

## 版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



# 相关性 搜索

利用Solr与Elasticsearch  
创建智能应用

Relevant Search: With applications for  
Solr and Elasticsearch



[美] Doug Turnbull 著  
John Berryman

Trey Grainger 作序

莫映 蔡宇飞 殷智勇 译



# 相关性搜索

利用Solr与Elasticsearch创建智能应用

Relevant Search : With applications for Solr and Elasticsearch

[美] Doug Turnbull 著  
John Berryman

Trey Grainger 作序

莫映 蔡宇飞 殷智勇 译

电子工业出版社  
Publishing House of Electronics Industry  
北京•BEIJING

## 内 容 简 介

本书揭开了相关性搜索的神秘面纱,告诉大家如何将 Elasticsearch 与 Solr 这样的搜索引擎作为可编程的相关性框架,从而表达业务排名规则。从这本书中你可学会如何结合各种外部数据源、分类方法以及文本分析手段对相关性进行编程,以满足用户的个性化需求,将令人满意的搜索结果呈现给用户。此外,相关性搜索也需要一定的软性技能。本书还将告诉读者怎样与业务人员协作,为业务找到正确的相关性需求,从而在搜索产品的整个研发生命周期内,实现相关性改进的良性循环。本书介绍了搜索引擎的基本原理,及相关性搜索的调试技术,用大量实例的方式详述了搜索引擎的诸多特性,以形成一整套针对相关性搜索的系统化方法,并倡导致力于提高搜索质量的企业文化。

本书适用于想利用 Elasticsearch 或 Solr 尝试构建智能搜索应用的开发人员。

Original English Language edition published by Manning Publications, USA. Copyright © 2016 by Manning Publications. Simplified Chinese-language edition copyright © 2017 by Publishing House of Electronics Industry. All rights reserved.

本书简体中文版专有出版权由 Manning Publications 授予电子工业出版社。未经许可,不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字:01-2016-7239

### 图书在版编目(CIP)数据

相关性搜索:利用Solr与Elasticsearch创建智能应用/(美)道格·特恩布尔(Doug Turnbull), (美)约翰·贝瑞曼(John Berryman)著;莫映,蔡宇飞,殷智勇译. —北京:电子工业出版社, 2017.10

书名原文: Relevant Search: With applications for Solr and Elasticsearch

ISBN 978-7-121-32721-6

I. ①相… II. ①道… ②约… ③莫… ④蔡… ⑤殷… III. ①搜索引擎—程序设计 IV. ①TP391.3

中国版本图书馆CIP数据核字(2017)第228614号

策划编辑:许 艳

责任编辑:刘 舫

印 刷:北京天宇星印刷厂

装 订:北京天宇星印刷厂

出版发行:电子工业出版社

北京市海淀区万寿路173信箱

邮编:100036

开 本:787×980 1/16

印张:24.5 字数:439千字

版 次:2017年10月第1版

印 次:2017年10月第1次印刷

定 价:99.00元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 [zltz@phei.com.cn](mailto:zltz@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式:010-51260888-819 [faq@phei.com.cn](mailto:faq@phei.com.cn)。

# 译者序

---

与本书结缘还要追溯到一年前。当时正值团队启动新产品的研发，需要一款查询性能优良的 NoSQL DB 作为数据存储方案。在考察了包括 Elasticsearch、Solr、Mongo、Cassandra 等一系列 NoSQL DB 之后，我们最终从实际需求出发，选择了与产品功能契合度更高的 Elasticsearch。于是大家开始了对 Elasticsearch 从零起步的探索。不过，在阅读了 Elasticsearch 的大量官方文档之后，大家发现，虽然通过文档的查阅可以了解 Elasticsearch 诸多特性的使用方法，但是这种工具书式的平铺直叙无法将知识有机地联系起来，形成系统而立体的认知。并且，在阅读官方文档的过程中我们也发现，自己对不少搜索相关的基础概念还不甚了解，于是只能借助于网络上搜到的一些支离破碎的快餐资源来补充营养。

为了解决这一问题，大家觉得团队成员们迫切需要一本良师益友式的专业书籍，它既能系统地介绍搜索的相关知识，又能结合当下流行的搜索引擎框架，做到理论与实践相结合。恰巧，电子工业出版社计算机出版分社的许艳老师联系到了我们，商讨一本刚从 Manning 出版社引进的外版书籍的翻译工作，该书正是以 Elasticsearch 和 Solr 为背景介绍相关性搜索的。而且，此书引进时刚刚面世不久，在亚马逊上甚至还没有开始售卖，因此所涉内容的时效性非常之高，正好是团队眼下急需的学习资源。于是，几位同事商量之后，觉得在学习之余，如果顺手将其译成中文，以惠及更多的业内同行，不失为一件利人利己的好事。故而，才有了读者眼前的这本中文版图书。

遇到本书是译者的幸运。书中围绕相关性搜索这一主题，全面系统地介绍了该领域的方方面面：从搜索引擎的基础知识，到相关性搜索的主要技术，再到各种高阶议题，直到当下前沿领域的研究成果，凡此种种，不一而足。两位作者通过朴实细腻的笔触，清晰无误的语言，循序渐进地将我们带入了相关性搜索的神奇世界。这里没有高深莫测的晦涩理论，只有生动有趣的示例讲解。值得一提的是，全书各



章所选的示例多以构建影片搜索应用这一任务为背景，一以贯之，精挑细选。通过来自 The Movie Database (TMDB) 的大量真实影片数据，为读者构建出了一个个实际可运行的搜索示例。其中，以经典系列影片“星际迷航”为主题的影片搜索应用，就在本书的前后多个章节中频频出现，足见作者构思精巧，用心良苦。读完本书，再读 Elasticsearch 或 Solr 的有关文档或书籍时，其中内容无一不有似曾相识的感觉；查阅其他介绍相关性搜索的文章，也有一种“一览众山小”的感觉。而面对现实生活中的各种应用，以及我们自己研发的产品，不禁让人联想，如果将书中所学应用其中，想必一定能画龙点睛，为之增色不少。

本书的翻译过程持续了将近十个月，中文版的字里行间都饱含了几位译者的辛勤汗水。回顾往昔，多艰之旅，历历在目，译稿最终得以成功付梓，实属不易。翻译过程中，从初译到终稿，每一章每一节基本都要经过反复推敲与琢磨至五六遍以上。因为是多人翻译，所以为了保证全书行文风格的统一，最后还进行了一次全面细致的统稿，几位译者都为此投入了极大的精力和时间。大家已经记不清有多少个日夜，当家人都已进入梦乡，自己却还在灯下埋首伏案；有多少个周末，把孩子托付给爱人照料，自己却在一旁奋笔疾书……

本书的翻译也是一次感恩之旅。感谢家人和朋友，没有他们一直以来的支持就不会有本书中文版的问世。感谢莫映的夫人李唯一女士，作为本书的首位读者，每每译稿新鲜出炉，都会经过她的耐心初校，以读者的视角为我们提出诸多中肯的修改建议。感谢智勇的家人，本书翻译之初正值智勇千金呱呱坠地，家人的理解和支持是这位新晋奶爸最大的前进动力。感谢宇飞的爱女，小小年纪就善解人意，能够体谅妈妈因为工作而少了与之相伴的时间。还要感谢博文视点的许艳老师，为我们牵线搭桥，感谢责编刘舫老师，为本书的后期审校尽心尽力。也要感谢我们这几位译者彼此间的相互扶持。大家利用各自的业余时间，以极大的热情投入到翻译工作中，默契配合，一路走来。当然，更应感谢本书的两位原作者 Doug Turnbull 先生和 John Berryman 先生，他们的睿智与经验成就了本书原作的好口碑。

最后，希望中文版的面世，不负原作的美誉，以及各位读者的厚望！

中文版译者

2017 年 8 月于北京，晴耕书斋



蔡宇飞

解智勇

# 推荐序

在过去十年里，搜索已经变得无处不在——关键字搜索框已经演变成查找数据和浏览大多数网站及应用的事实上的标准用户界面。与此同时，对大多数组织来说，若非被严重忽视，要想提供真正具有相关性的搜索体验一直以来都绝非易事。

强大的开源技术已经能做到在几乎零编码的情况下（如 Apache Solr 和 Elasticsearch），以分布式的、高度可伸缩的方式，实现高效运行和功能丰富的搜索（如 Apache Lucene）。这为几乎所有的开发人员在大数据时代建立起一个“在一般意义上相关（generally relevant）”的实时搜索引擎提供了必要的基础架构。随着搜索在基础架构方面有越来越多的难题得到了解决，加之解决方案的商品化进程，竞争的差异已经从如何提供快速、可伸缩的搜索，转变成如何针对用户的信息需求提供最为相关的匹配。换言之，提供“在一般意义上相关”的结果已经远远不够了——谷歌以及其他顶级的搜索引擎现在已经把用户培养成为这样一种群体，他们期望搜索应用几乎能读懂自己内心的想法。本书所讨论的，就是我们如何更加积极地朝着理解用户意图的方向去努力。

Doug Turnbull 和 John Berryman 是两位经验丰富的搜索和相关性领域的专家，我认识他们已经很多年了，大家时常会在出席搜索大会时遇到。我还能回忆起与他们一起讨论的美好时光，我们共同探讨了如何解决搜索相关性、推荐和个性化方面的一些世界级难题。没有人会比我更加欣喜地看到他们将自己独特的专业知识融入这本书中——这是我读过的最好的、最引人入胜的技术书籍之一。

相关性调优是一大难题——它经常被误解，而且当出现错误时，往往不会立即显现出来。为了识别出有问题的模式，我们通常需要看到许多错误的样例才行，并且在没有真正看到结果显示的情况下，要想知道什么才是更好的调优结果往往很困难。遗憾的是，通常一个搜索系统直到被部署到生产环境之后，组织才开始意识到

默认提供的相关性和真实的、受领域驱动的、个性化匹配之间存在的差距。

不仅如此，那些处理相关性所需的技能（如领域专长、特征工程、机器学习、本体理论、用户测试、自然语言处理等）与那些构建和维护可扩展的基础架构所需的技能（如分布式系统、数据结构、性能和并发、硬件利用率、网络传输和通信等）也是非常不同的。相关性技术工程师这一角色在许多组织中几乎完全缺失，从而给我们留下了许多未曾发掘的潜力，可以营造出真正让用户满意、并显著推动公司发展的搜索体验。

从手动输入关键字进行搜索到完全自动化的推荐，这一范围内的各种不同程度的个性化，也给我们带来了许多机会，可以为每一位用户的特定需求提供相关性匹配。本书作者相当出色地解释了对搜索特征或信号进行建模时各种方法间的细微差别，从而充分利用了这一范围内的各种不同程度的个性化。掌握了本书中介绍的技术，我们就可以很好地胜任相关性技术工程师的角色，并能够解决在创建真正个性化的相关搜索体验时遇到的许多最具挑战性的问题。

Trey Grainger

*Solr in Action* 一书的作者

Lucidworks 技术部高级副总裁

# 前言

---

John 和我是在共同为 OpenSource Connections (OSC) 做咨询工作、帮助客户解决棘手的搜索问题时认识的。我们有时一起诊断性能问题（好让系统跑得更快），有时帮助构建搜索应用。所有项目都有一系列衡量成功与否的简单指标：系统运行更快了吗？应用开发完成了吗？

但是，搜索相关性并不遵循这些规则。而且从谷歌时代成长起来的用户是不会容忍“还算凑合”这样的搜索的。他们想要的是“绝顶聪明”的搜索。他们希望搜索能够优先考虑其所关心的条件标准，而不是像搜索引擎通常那样，盲目地去猜测相关性。

就像飞蛾被火焰所吸引一样，我们都被这一难题深深吸引了。而且正如所谓的飞蛾那样，我们也常常发现自己是在“扑火”。经过这些惨痛的教训，我们坚持了下来并且得到了成长，在我们最初认为极其困难的任务上取得了成功。

在此期间，我们在 OSC 的博客上也看到了同样的心声。我们意识到有关搜索相关性的问题被记录下来的文字少之又少。于是，我们提出了诸如测试驱动相关性（test driven relevancy）这样的观点。我们记录下了自己心中的困惑、遇到的问题，以及取得的成功。我们一起试验了机器学习的各种方法，比如潜在语义分析（latent semantic analysis）。我们研究了 Lucene 的内部机制并探索了通过构建自定义搜索组件来解决实际问题的相关技术。我们还开始了对信息检索的研究。随着我们所掌握的解决疑难问题的技术越来越多，我们也持续不断地将它们记录为文字。

然而，博客有其自身的局限性。John 和我一直希望以书的形式更加系统地阐明我们的观点。幸运的是，我们经历了一连串有趣的事情，机会往往就会自动找上门来。我在一个本地的技术交流会上与 Andrew Montalenti 一起做了一个关于 Python 并发的演讲。因为 Andrew 在 PyCon 上做过这次演讲，Manning 就给 Andrew 打电话来



# 致谢

---

在开始撰写本书的几周前，我们两家都迎来了小宝宝。我们要把最诚挚的感谢和爱意送给我们的爱人，Khara Turnbull 和 Kumiko Berryman。我们把连续数个周末的时间都用来写书，而她们也都坚持了下来——在此期间，Khara 完成了她自己的一本书，Kumiko 成功地经历了一次长途越野和房屋出售。现在是时候放个长假了！

本书的成功付梓也离不开 OpenSource Connections 的创始人 Eric Pugh。作为我们的“老板”，是他把我们推到了写作、演讲和解惑的聚光灯下。作为一位领导者，Eric 能够让你的热情成为他的热情。如果不是 Eric 摘掉“辅轮”（有时甚至坚持“独轮”），我们就不会意识到，自己竟然能胜任写作或解惑的工作。Eric 告诉我们，每个人都可以成为思想领袖，包括我们自己。

感谢 TMDB 提供的数据和支持。我们曾经花费大把的时间试图找到理想的数据集。TMDB (<http://themoviedb.org>) 不仅提供了丰富的搜索数据集，而且在我们遇到程序错误和问题的時候（通常是我们自己的代码里的错误），TMDB 也能为我们以及我们的早期读者提供支持。特别要感谢的是 Travis Bell，他总是及时回复我们的问题和邮件。

写书是一项团队活动，我们要感谢 Manning 出版社本书制作团队中的每一位成员：Marina Michaels，我们的开发编辑；Aaron Colcord，技术开发编辑；Valentin Crettaz，技术校对；Frank Pohlmann 和 Mike Stephens，策划编辑；还有负责营销的 Candace Gillhoolley。

我们也要感谢很多参与审稿的朋友，他们阅读了本书最初的书稿，并提出了许多有益的建议，包括 John Guthrie，Martin Beer，Arthur Zubarev，Elman Krinker，Amit Lamba，Marc-Oliver Scheele，Ian Stirk，Joseph Wang，Stuart Woodward，Ursin Stauss，Russ Cam，Michael Fink，Gregor Zurowski，Dimitrios Kouzis-Loukas，

Jeremy Gailor 和 Keith Webster。

另外要感谢 Andrew Montalenti, 他为我们与 Manning 建立了联系。还要感谢 Shay Banon 的帮助, 他是 Elasticsearch 的创始人, 坦率地说, 他是一个很和蔼的人。感谢我们的同事, Trey Grainger, Matt Overstreet, Rena Morse, David Smiley, Grant Ingersoll, Yonik Seeley, Rene Kriegler, Peter Dixon-Moses, Charlie Hull 和 Drew Farris, 感谢这些年来与我们在搜索和相关性方面的这么多精彩讨论。还要特别感谢 Trey, 他为我们这本书写了推荐序。

感谢每一位家人对我们的支持。尤其是我们的孩子们: Megume Berryman, Ian Turnbull 和 Murray Turnbull。感谢我们在 OpenSource Connections 和 Eventbrite 上的“工作大家庭”, 让我们能够把大量精力投入到本书的写作上。

# 关于本书

---

本书将告诉大家，在响应用户的搜索时要给出用户满意和认可的内容。我们将学习如何根据搜索条件，而不是对搜索引擎的神秘猜测，来严格控制对搜索结果的排名。我们会简要介绍深入定制 Solr 或 Elasticsearch 相关性排名的方法，以及如何采取措施帮助大家发掘相关性对应用而言的意义。

## 谁应该阅读本书

本书的目标读者是那些渴望了解为什么搜索引擎无法“领会”用户搜索意图的 Solr 或 Elasticsearch 开发人员。对搜索引擎至少有基本了解的读者，可以通过本书将他们的技能提升到更高的层次。虽然这是一本技术方面的书籍，但从组织机构和产品战略的角度来看，它的大部分内容都是围绕相关性展开的，因此也适合于产品经理、内容战略的制订者、市场营销人员，或专注搜索的领域专家阅读。

## 本书是如何组织的

本书首先介绍了相关技术的基础知识，然后逐步上升到定义和解决搜索相关问题时我们所要面对的产品策略和文化议题，最后介绍了如何实施个性化搜索、语义搜索以及推荐。

第 1 章从讨论相关性问题开始。问题涉及的领域包括网络搜索、电子商务，以及专家搜索等。本章讨论了学术界对于我们在相关性领域所做的尝试都提供了什么样的支持。最后，我们简要介绍了本书在解决相关问题时所采取的技术策略。

第 2 章对 Lucene 的核心数据结构及其算法做了快速回顾，因为它们与相关性是紧密关联的。我们将会看到，为了寻找相关性内容，基于 Lucene 的搜索为我们提供

了一个如此令人难以置信的框架。

第3章告诉大家如何对相关性进行调试。在第2章介绍的数据结构和算法不起作用的时候，我们就需要拿出自己的“工具箱”，弄清楚搜索在哪里出了问题。

第4章展示了如何利用搜索引擎的分析流程将内容和搜索分解成可描述的特征。这一基本技巧可以让我们学会如何利用分析手段让所有内容都能被找到。

第5章开始讨论针对多个字段的查询策略。在本章中，我们会告诉大家如何构造查询语句，有针对性地去度量那些在搜索阶段对用户而言至关重要的排名因素。

第6章继续我们有关查询策略的讨论。在这一章我们重点关注的是以词为中心（term-centric）的技术，以及相应的搜索策略，以支持用户对相关性的朴素理解。

第7章为大家展示了评价调整（score-shaping）技术，比如放大（boosting）和过滤（filtering）。我们时常需要突出近期产生的内容、高利润率的产品，或者距离较近的位置，以此来对搜索加以控制。

第8章为大家展示了一系列可以帮助用户找到相关性内容的替代方法。有时，当相关性排名效果不佳的时候，一些UI组件，比如可供浏览的切面（browsable facets）、自动补全（autocomplete），以及高亮显示（highlighting），也许用这些方法将用户引入正途更为简单。

第9章我们构建了一个完整的以相关性为目标的搜索应用，本章将会为大家奉上具有专业眼光的Yowling。既然已经掌握了相关性技术工程师所具备的一系列技能，本章我们将从头至尾见证一次完整的产品开发流程。

第10章从产品战略的角度上升到了一个更高的层面，其目的在于关注文化和组织机构方面的一些因素。一个关注搜索的组织是如何确定何谓相关的呢？我们会看到，一个组织必须实现快速而准确的反馈回路，才能正确引导相关性技术工程师的研发工作。

第11章将我们的视野拓展到了搜索引擎以外的地方。本章会向大家介绍机器学习、个性化搜索，以及语义搜索是如何协同工作，一起来提高搜索引擎的相关性排名的。

附录A带领大家按照我们曾经走过的流程，利用The Movie Database（TMDB）API一步步将本书所用的数据载入Elasticsearch中。

附录B通过对照Elasticsearch和Solr之间的相关性功能，指导Solr读者阅读本书。

## 关于代码

本书包含了许多源代码的例子，形式包括带编号的清单，以及普通的文本行。对于这两种情况，源代码都以等宽字体进行了格式化，从而将其与普通文本进行区分。有时代码也会以粗体显示，目的是为了突出相对于本章前面步骤的变化，比如当一个新功能被加入已有的代码行时。

在许多情况下，最初的源代码都已经被重新进行了格式化处理；我们加入了换行并修改了缩进，目的是为了能够适应本书的可用版面。另外，当我们在文中对代码做了解释之后，源代码中的注释通常会被从清单中移除。许多代码清单都会伴有一定的注解，旨在突出显示某些重要的概念。

这些例子都已在 Elasticsearch 2.0 和 Python 2.7 下测试通过。

大家可以在 Manning 的网站 ([www.manning.com/books/relevant-search](http://www.manning.com/books/relevant-search)) 上以及本书的 GitHub 库 (<http://github.com/o19s/relevant-search-book>) 中找到第 3 章至第 9 章的代码。为了便于试验，这些例子都是用 iPython Notebook/Jupyter 编写的。README 文件详细说明了运行代码所需的准备工作。

## 作者在线

购买本书的读者可以免费访问一个由 Manning 出版社运作的私有论坛，在那里你可以对本书发表评论、询问技术问题，并得到作者和其他用户的帮助。要访问和订阅该论坛，请在浏览器中打开 [www.manning.com/books/relevant-search](http://www.manning.com/books/relevant-search)。该网页提供的信息包括：如何在成功注册之后加入论坛，你可以得到什么样的帮助，以及论坛内的行为规范。

Manning 出版社承诺为读者提供这样一个场所，在那里不同读者之间，以及读者和作者之间可以建立起有意义的对话。本书作者并不承诺任何具体程度的参与，他们对本书论坛的贡献是自愿的（无偿的）。我们建议大家试着问一些具有挑战性的问题，以激起他们的兴趣！

本书一经出版，就可以通过出版社的网站访问作者在线论坛和以往讨论的存档。

## 其他在线资源

如果你想了解更多信息，我们推荐以下质量不错的资源。

- OpenSource Connection 的博客 (<http://opensourceconnections.com/blog>)。
- John Berryman 的个人博客 (<http://thoughtbox.solutions>)。
- Elastic 的博客 ([www.elastic.co/blog](http://www.elastic.co/blog))。
- Lucidwork 的博客 (<https://lucidworks.com/blog>)。
- Salmon Run, Sujit Pal 的 Solr 博客 (<http://sujitpal.blogspot.com/>)。
- Solr Start 的简讯 ([www.solr-start.com](http://www.solr-start.com))。

有关搜索和信息检索方面更为一般性的讨论，我们建议参考下面这部宝典：

- 由 Christopher Manning 等人编写的 *Introduction to Information Retrieval*, (剑桥大学出版社, 2008), <http://nlp.stanford.edu/IR-book/>。

有关 Solr/Elasticsearch 的具体问题，我们建议访问各自的技术论坛：

- <http://discuss.elastic.co>。
- <http://lucene.apache.org/solr/resources.html>。

## 读者服务

轻松注册成为博文视点社区用户 ([www.broadview.com.cn](http://www.broadview.com.cn))，扫码直达本书页面。

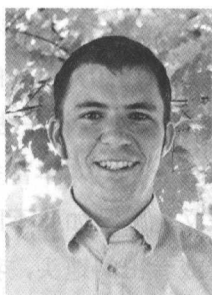
- 提交勘误：你对书中内容的修改意见可在[提交勘误](#)处提交，若被采纳，将获赠博文视点社区积分（在你购买电子书时，积分可用来抵扣相应金额）。
- 交流互动：在页面下方[读者评论](#)处留下你的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/32721>



# 关于作者

---



**Doug Turnbull** 在 OpenSource Connections 上领导着一项搜索相关性的咨询业务，在那里他经常发表观点和更新博客。Doug 利用各种搜索和自然语言处理技术（NLP）为多个领域的客户构建语义丰富的相关性搜索体验。



**John Berryman** 的第一份职业是航空工程师，但在航空领域工作了几年之后，他发现编写程序或解决数学难题才是他最喜欢的工作。最终，John 撇下了飞机和卫星，开始全职工作于软件开发、基础架构，以及搜索技术领域。目前，John 供职于 Eventbrite，帮助利用 Elasticsearch 构建事件活动的发现、搜索及推荐。

# 关于封面插图

---

本书的封面人物名叫“Homme de l'Isle de Pathmos”，他是来自希腊帕特莫斯岛的一个男人。该图取自 Jacques Grasset de Saint-Sauveur（1757–1810）所著的一本描绘来自不同国家服饰的画册，名为 *Costumes de Différents Pays*，于 1797 年在法国出版。每幅插图都是手工绘制并着色而成的。Grasset de Saint-Sauveur 的这本丰富多彩的画册向我们生动地展示了 200 年前世界各地的城镇和地区在文化上是如此不同。人们彼此之间相互阻隔，说着不同的方言和俚语。在大街小巷或者田间地头，我们很容易就能根据人们的衣着辨别出他们住在什么地方，做些什么买卖，或者身份地位如何。

从那以后，我们的衣着方式发生了改变，那时如此丰富的地域多样性也渐渐消失。现在人们已经很难区分出不同大洲的居民，更不用说不同的城镇、地区或国家了。也许我们已经将文化的多样性换成了一种更加多元化的个人生活——当然是一种更加多元和快节奏的科技生活。

当人们难以分辨不同的计算机书籍时，Manning 根据两个世纪前不同地域的人们在生活上丰富的多样性来设计书籍的封面，以此来颂扬计算机领域里的发明创新和开拓进取，通过 Grasset de Saint-Sauveur 的图片把人们带回到当初的生活。



# 目录

第1章 搜索的相关性问题 .....	1
1.1 我们的目标：掌握相关性技术研发的技能 .....	2
1.2 为什么搜索的相关性如此之难 .....	3
1.2.1 什么是具备“相关性”的搜索结果 .....	4
1.2.2 搜索：没有银弹 .....	6
1.3 来自相关性研究的启示 .....	7
1.3.1 信息检索 .....	7
1.3.2 能否利用信息检索解决相关性问题的 .....	9
1.4 如何解决相关性 .....	11
1.5 不只是技术：管理、协作与反馈 .....	13
1.6 本章小结 .....	16
第2章 搜索——幕后揭秘 .....	17
2.1 搜索101 .....	18
2.1.1 什么是搜索文档 .....	19

2.1.2	对内容进行搜索	19
2.1.3	通过搜索来探索内容	21
2.1.4	获取进入搜索引擎的内容	21
2.2	搜索引擎的数据结构	23
2.2.1	倒排索引	23
2.2.2	倒排索引的其他内容	25
2.3	对内容进行索引：提取、充实、分析和索引	26
2.3.1	将内容提取为文档	28
2.3.2	充实文档以清理、强化与合并数据	28
2.3.3	执行分析	29
2.3.4	索引	33
2.4	文档的搜索和获取	34
2.4.1	布尔搜索：AND/OR/NOT	34
2.4.2	基于 Lucene 搜索的布尔查询（MUST/MUST_NOT/SHOULD）	36
2.4.3	位置和短语匹配	37
2.4.4	助力用户浏览：过滤、切面和聚合	38
2.4.5	排序、结果排名，以及相关性	39
2.5	本章小结	42
第3章	调试我们的第一个相关性问题	43
3.1	Solr和Elasticsearch的应用：基于Elasticsearch的例子	44
3.2	最了不起的数据集：TMDB	45
3.3	用Python语言编写的例子	46
3.4	第一个搜索应用	46
3.4.1	针对 TMDB Elasticsearch 索引的第一次搜索	49
3.5	调试查询匹配	52
3.5.1	检查底层查询策略	53
3.5.2	剖析查询解析	54
3.5.3	调试分析，解决匹配问题	55
3.5.4	比较查询条件和倒排索引	58
3.5.5	通过修改分析器来修正我们的匹配	59

3.6	调试排名	62
3.6.1	利用 Lucene 的解释功能来剖析相关性评价	63
3.6.2	向量空间模型、相关性解释信息和我们	68
3.6.3	向量空间模型在实践中的注意事项	71
3.6.4	通过对匹配的评价来度量相关性	72
3.6.5	用 $TF \times IDF$ 计算权重	74
3.6.6	谎言、该死的谎言和相似度	75
3.6.7	决定搜索词重要性的因素	77
3.6.8	解决 Space Jam 和 alien 的排名问题	78
3.7	问题解决了? 工作永远做不完!	80
3.8	本章小结	81
<b>第4章</b>	<b>驾驭token</b>	<b>83</b>
4.1	将token作为文档特征	84
4.1.1	匹配的流程	85
4.1.2	token, 不只是单词	85
4.2	控制查准率和查全率	86
4.2.1	查准率和查全率的例子	86
4.2.2	查准率或查全率的分析	89
4.2.3	一味提高查全率	93
4.3	查准率和查全率——让鱼和熊掌兼得	95
4.3.1	评价单一字段中特征的强度	95
4.3.2	超越 $TF \times IDF$ 的评价: 多搜索词与多字段	99
4.4	分析策略	100
4.4.1	处理分隔符	100
4.4.2	捕获同义词的语义	103
4.4.3	在搜索中为专指性建模	107
4.4.4	利用同义词为专指性建模	107
4.4.5	利用路径为专指性建模	110
4.4.6	对整个世界分词	112
4.4.7	对整数分词	112

4.4.8	对地理数据分词	113
4.4.9	对歌曲分词	115
4.5	本章小结	118
<b>第5章</b>	<b>多字段搜索基础</b>	<b>119</b>
5.1	信号及信号建模	121
5.1.1	什么是信号	121
5.1.2	从源数据模型开始	122
5.1.3	实现信号	125
5.1.4	信号建模：为数据的相关性建模	126
5.2	TMDB——搜索，人类最后的边疆	127
5.2.1	违反基本法则	129
5.2.2	让嵌套文档扁平化	129
5.3	在以字段为中心的搜索中给信号建模	132
5.3.1	从 best_fields 开始	136
5.3.2	控制搜索结果中的字段偏好	139
5.3.3	可以使用信号更精准的 best_fields 吗	141
5.3.4	让失败者分享荣耀：为 best_fields 校准	144
5.3.5	利用 most_fields 统计多个信号	147
5.3.6	在 most_fields 中缩放信号	149
5.3.7	什么时候其他匹配才无关紧要	151
5.3.8	有关 most_fields 的结论是什么	152
5.4	本章小结	153
<b>第6章</b>	<b>以词为中心的搜索</b>	<b>154</b>
6.1	什么是以词为中心的搜索	155
6.2	我们为什么需要以词为中心的搜索	157
6.2.1	猎寻“白化象”	157
6.2.2	在“星际迷航”的例子中寻找白化象问题	160
6.2.3	避免信号冲突	162

6.2.4	理解信号冲突的机理	163
6.3	完成第一个以词为中心的搜索	165
6.3.1	使用以词为中心的排名函数	166
6.3.2	运行以词为中心的查询解析器(深入底层)	169
6.3.3	理解字段同步	170
6.3.4	字段同步和信号建模	171
6.3.5	查询解析器和信号冲突	172
6.3.6	对以词为中心的搜索进行调优	174
6.4	在以词为中心的搜索中解决信号冲突	176
6.4.1	将字段合并成自定义全字段	177
6.4.2	利用 cross_fields 解决信号冲突	181
6.5	结合以字段为中心和以词为中心的策略:鱼与熊掌兼得	183
6.5.1	将“相似字段”分到一组	183
6.5.2	理解相似字段的局限	185
6.5.3	将贪婪的简单搜索和保守的放大器结合起来	186
6.5.4	以词为中心与以字段为中心,查准率与查全率	189
6.5.5	考虑过滤、放大,以及重新排名	190
6.6	本章小结	190
<b>第7章 调整相关性函数</b>		<b>192</b>
7.1	何谓评价调整	193
7.2	放大:通过突出结果来实现调整	195
7.2.1	放大:最后的边疆	195
7.2.2	放大时——选择加法运算还是乘法运算, 布尔查询还是函数查询?	197
7.2.3	选择第一扇门:利用布尔查询进行加法放大	199
7.2.4	选择第二扇门:利用数学运算进行排名的函数查询	202
7.2.5	函数查询实践:简单的乘法放大	204
7.2.6	放大处理的基础:信号,处处是信号	206
7.3	过滤:通过排除的方法对结果进行调整	206
7.4	满足业务需求的评价调整策略	208

7.4.1	搜索所有影片 .....	209
7.4.2	对放大信号进行建模 .....	211
7.4.3	构造排名函数：增加具有较高价值的层级 .....	215
7.4.4	利用函数查询对具有较高价值的层级进行评价 .....	219
7.4.5	忽略 $TF \times IDF$ .....	221
7.4.6	捕捉综合质量指标 .....	222
7.4.7	达成用户的时效性目标 .....	224
7.4.8	结合函数查询 .....	227
7.4.9	把一切联系起来 .....	230
7.5	本章小结 .....	230
<b>第8章</b>	<b>提供相关性反馈 .....</b>	<b>232</b>
8.1	搜索框中的相关性反馈 .....	234
8.1.1	利用“即输即搜”提供即时结果 .....	234
8.1.2	利用“搜索补全”帮助用户找到最佳查询 .....	236
8.1.3	利用搜索建议来修正输入和拼写错误 .....	244
8.2	浏览期间的相关性反馈 .....	247
8.2.1	构建基于切面的浏览 .....	249
8.2.2	提供面包线导航 .....	251
8.2.3	选择其他的结果排序方式 .....	252
8.3	搜索结果清单中的相关性反馈 .....	253
8.3.1	什么信息应该出现在搜索结果中 .....	254
8.3.2	通过文本片段与高亮提供相关性反馈 .....	255
8.3.3	对相似文档分组 .....	259
8.3.4	在用户搜不到结果时给予帮助 .....	261
8.4	本章小结 .....	262
<b>第9章</b>	<b>设计以相关性为核心的搜索应用 .....</b>	<b>263</b>
9.1	Yowl! 一个绝佳的新起点 .....	264
9.2	信息和需求的收集 .....	265

9.2.1 理解用户及其信息需求.....	265
9.2.2 理解业务需求.....	267
9.2.3 找出必要及可用的信息.....	268
9.3 搜索应用的设计.....	269
9.3.1 将用户体验可视化.....	270
9.3.2 定义字段和模型的信号.....	273
9.3.3 信号的组合与平衡.....	274
9.4 部署、监控和改进.....	286
9.4.1 监控.....	286
9.4.2 找出问题并解决它们.....	288
9.5 知道什么是恰到好处.....	289
9.6 本章小结.....	290
<b>第10章 以相关性为核心的企业.....</b>	<b>292</b>
10.1 反馈：以相关性为核心的企业所依赖的基石.....	294
10.2 为什么以用户为中心的文化比数据驱动的文化更重要.....	296
10.3 无视相关性的天马行空.....	299
10.4 相关性反馈的觉醒：领域专家和专业用户.....	301
10.5 相关性反馈的成长：内容管理.....	303
10.5.1 内容管理员的角色.....	304
10.5.2 与内容管理员缺乏交流的风险.....	306
10.6 让相关性更加流畅：工程师/内容管理员的结对.....	307
10.7 让相关性加速：测试驱动的相关性.....	309
10.7.1 理解测试驱动的相关性.....	309
10.7.2 使用带用户行为数据的测试驱动相关性.....	312
10.8 超越测试驱动的相关性：学习排序.....	313
10.9 本章小结.....	315
<b>第11章 语义和个性化搜索.....</b>	<b>316</b>
11.1 基于用户概况的个性化搜索.....	318

11.1.1	收集用户的概况信息.....	319
11.1.2	将概要信息与文档索引紧密关联.....	319
11.2	基于用户行为的个性化搜索.....	320
11.2.1	引入协同过滤.....	321
11.2.2	使用共现计数的基本协同过滤算法.....	322
11.2.3	将用户行为信息与文档索引紧密关联.....	327
11.3	构建概念性搜索的基本方法.....	332
11.3.1	构建概念性信号.....	333
11.3.2	利用同义词对内容进行扩充.....	334
11.4	利用机器学习来构建概念性搜索.....	336
11.4.1	概念性搜索中短语的重要性.....	338
11.5	连接个性化搜索与概念性搜索.....	338
11.6	推荐是一种广义的搜索.....	339
11.6.1	用推荐代替搜索.....	341
11.7	祝愿大家有一个美好的相关性搜索之旅.....	342
11.8	本章小结.....	343
附录A	直接根据TMDB建立索引.....	344
附录B	Solr读者指南.....	351



# 搜索的相关性问题

## 本章要点

- 搜索无处不在
- 打造相关性搜索体验的挑战
- 这一挑战在重要搜索领域中的例子
- 现成的解决方案在解决这类问题时捉襟见肘
- 本书实现相关性搜索采用的方法

想让搜索引擎工作正常可能是一件令人抓狂的事。无论你是刚开始使用 Solr 或 Elasticsearch 的新手，还是具备多年经验的熟手，都有可能在质量低劣的搜索结果里苦苦挣扎过。现成的配置已经满足不了需求，为了让相关性搜索结果能有哪怕一点点改进，你都需要使出浑身解数。

一提到相关性排名，搜索引擎就有点像一个神秘的黑盒。人们试图忽略相关问题，把焦点从搜索转移到应用中那些更显而易见的地方，比如性能或用户界面。但不幸的是，搜索的相关性排名是一项绕不过去的工作。在如今的各种应用中，用户处理大量内容的需求日益增长。不管这些内容是商品、书籍、日志信息、电子邮件、房屋短租、还是医学论文——搜索框总是用户探寻答案的首选。如果没有直观的搜

索功能，以通俗易懂的形式给出问题的答案，用户就会彻底抓瞎。因此，不管搜索有多么令人抓狂，看起来有多么神秘，我们都必须找到解决的办法。

相关性搜索（relevant search）技术揭开了相关性的神秘面纱。到底什么是相关性呢？它处于搜索引擎价值主张的核心地位。所谓相关性，就是根据内容对用户及业务需求的满足程度，对搜索内容进行排名的一门学问。细节决定成败：搜索结果排名针对的是什么样的内容？（是 tweets？商品？还是豆豆公仔？）什么样的用户？（是医生？还是精通技术的顾客？）什么样的门类？（是日文搜索？日用百货？还是法律专业信息？）用户期望得到什么？（是购物经验？还是图书馆的卡片目录？）以及雇主期望从这次交互中得到什么？（是利润？流量？还是商誉？）在我们的各种应用中，搜索已经变成一项无处不在的功能，默默运转而不张扬。回答了上述这些问题（即准确理解了相关性）就意味着用户体验的增进，反之则意味着用户体验的下降。

## 1.1 我们的目标：掌握相关性技术研发的技能

如何才能做到这一点呢？相关性搜索这项技术将告诉我们，要成为一名相关性技术研发工程师所要掌握的各项技能。它会将搜索引擎打造成一个看似智能的系统，能够理解用户和业务的需求。为此，我们需要让搜索引擎理解内容中包含的重要特征（feature）：比如餐厅所处的位置，书中出现的词汇，或者衬衫印染的颜色这样的属性。有了准确的特征，我们就可以在用户搜索时，计算出什么对他们而言是重要的：这家餐厅离我有多远？这本书上讲的内容对我有帮助吗？这件衬衫和我刚买的裤子搭配吗？这些在搜索期间影响排名的因素，是用来衡量用户所关心的内容的，我们称之为信号（signals）。大家会看到，我们时常面临的挑战，就是在不同特征之间做出选择，并实现能满足用户与业务需求的信号。

但是技术把戏只是工作的一部分（如图 1.1 所示）。明白要做什么可能比知道怎么做更重要。具有讽刺意味的是，相关性技术工程师很少了解“相关性”在某个具体应用里的含义。相反，倒是另一拨人——通常是那些不懂技术的同事们——懂得内容、业务，以及用户目标。我们将学习，在一家视相关性为核心的企业里，如何倡导和利用更广泛的业务知识及用户行为数据，来揭示用户希望从搜索中获得的体验。

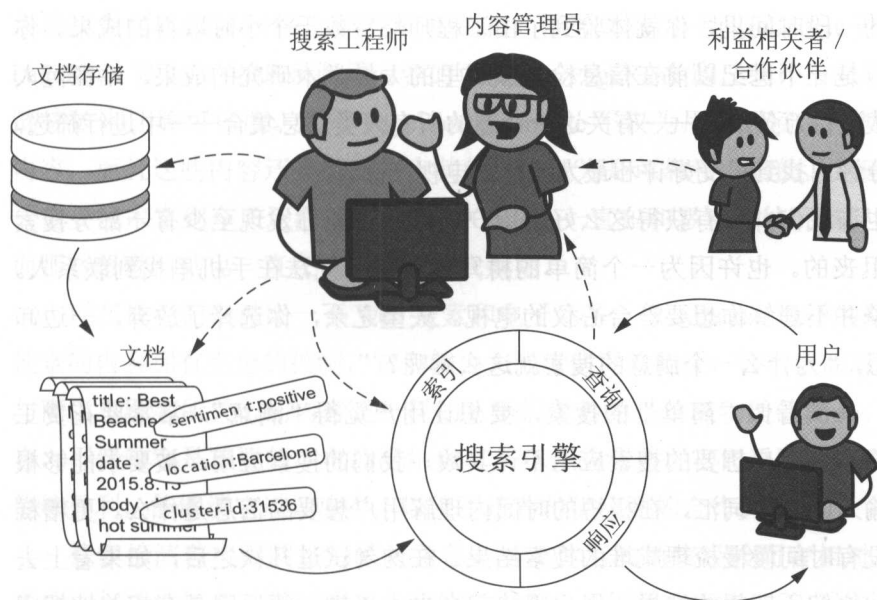


图 1.1 相关性技术工程师利用搜索引擎和后端技术来表达业务排名逻辑。他们就相关性事宜与跨职能团队紧密协作，并通过对用户的度量获得大量的反馈信息。

在本章的后面（以及整本书中），我们将详细阐述这些概念。但是为了奠定一个良好的基础，我们会在本章的剩余部分对相关性问题进行定义。为什么解决相关性这么难？为了解决这个问题人们尝试过哪些手段？随后我们再来简要介绍一下本书在解决相关性时所采用的方法。

## 1.2 为什么搜索的相关性如此之难

搜索的相关性问题如此之难，某种程度上是由于我们对搜索行为的轻视。搜索应用执行用户的搜索查询（即输入到搜索框中的文本），并根据内容匹配的情况，尝试对其进行排名。

这一动作发生得如此频繁，以至于很少被人察觉。想想我们自己的经验。你也许早上醒来，煮了一杯咖啡，开始摆弄你的智能手机。读读新闻，逛逛 Facebook，看看邮箱。甚至在咖啡煮好之前，你已经不假思索地和十几个搜索应用打过交道了。你有没有在手机的联系人列表里找过好友，给他发过消息？有没有搜索过一封重要的电子邮件？有没有和 Siri 聊过天？有没有为了满足自己的好奇心在 Google 上搜索过什么？有没有在 Amazon 上淘过梦寐以求的 50 英寸平板电视？

在短短的一段时间里，你就体验到了由工程师耗费数千个小时取得的成果。你所接触到的，是一个世纪以前在信息检索领域里的大量学术研究的成果。站在巨人的肩膀上，从数百万条信息——有关这一话题的所有人类信息集合——中进行筛选，并在区区几分钟内找到最受好评和最为流行的电视产品。

或者，也许我们并没有获得这么好的用户体验。可能你发现至少有一部分搜索体验是令人沮丧的。也许因为一个简单的拼写错误，你无法在手机里找到联系人。也许搜索引擎并不理解你想要一台心仪的电视。失望之余，你选择了放弃，一边卸载应用一边想，“为什么一个满意的搜索就这么难呢？”

事实上，一次看似“简单”的搜索，要想让用户觉得“满意”，常常要花费工程师大量的努力。用户想要的搜索应用不计其数。我们的搜索应用，被要求能够根据用户匆匆输入的几个词汇，在极短的时间内理解用户想要的信息是什么。更糟糕的是，用户没有时间慢慢梳理成堆的搜索结果。在匆匆试过几次之后，如果看上去搜索无法返回他们所期望的结果，用户很快就会失去兴趣。而返回具有相关性搜索结果的时间窗口也非常之短，且总是被一再压缩。

也许你会想，“问题看起来固然很难，但为什么我们无法轻易解决它呢？”搜索技术的出现已经有些日子了；像 Solr 或 Elasticsearch 这样的搜索引擎难道不应该总为我们返回正确的结果吗？或者为什么不让用户去用 Google 呢？为什么像 Amazon A9 这样现成的商业解决方案无法解决我们的搜索问题呢？

### 1.2.1 什么是具备“相关性”的搜索结果

我们很容易误以为搜索是一个单一问题。事实上，不同的搜索应用之间，彼此差异非常大。诚然，一个典型的搜索应用会让用户输入文本，然后对文档进行过滤，并将一组经过排序的结果反馈给用户。但不要被表象所迷惑。每个应用都有截然不同的相关性预期。让我们来看几个常见类型的搜索应用，从而明白你的应用也许有它自己对相关性的独特定义。

首先，我们来看看 Web 搜索。随着 Web 的发展，早期的 Web 搜索引擎很容易被恶意站点欺骗。那些建立恶意站点的人将一些词汇混入他们的网页里，以此来误导搜索引擎。最好的情况是，这些搜索引擎为用户查询返回了过时的信息。最糟的情况是，它们会把用户导向垃圾或恶意网页。

Google 认识到 Web 的相关性要依靠信任，而不是只依靠文本。给用户提供的

帮助，需要过滤掉网络上那些不可靠的“害群之马”。所以 Google 开发出了它的 PageRank 算法<sup>1</sup>，以此来衡量内容的可信度。PageRank 通过统计指向某站点的 Web 链接数来计算可信度。利用 PageRank，Google 不仅为用户返回了与搜索相匹配的内容，而且这些内容还是被网络上其他地方认为是可靠和可信的。返回可信内容这一重要技术直到今天还在使用，因为 Google 一直在和恶意网站玩猫捉老鼠的游戏，而那些恶意网站也一直在试图打破游戏规则。

现在让我们来对比一下 Web 搜索和电子商务。像 Amazon 这样的网站，对于被搜索的内容拥有完全的控制，它们较少关注可信度。相反，针对电商用户的相关性，与商店里顾客所关心的内容是一样的：他们关心的是价廉物美的商品。但对商店而言，不仅顾客是重要的，电商网站也有其自身的利益。它们返回的搜索结果也必须也能够盈利，能够清空过期库存，以及不违背与供应商的合作关系。

搜索扮演着电商网站的销售角色。对实体店的销售经验而言相对重要的因素，也应由相关性技术工程师放入电商搜索中去。工程师希望能打造出这样一款搜索应用，它能急人之所需，让顾客乘兴而归。对电子商务而言，相关性不仅意味着提高用户的消费满意度，还意味着能让电商盈利。

还有另一种搜索，多见于医疗、法律和研究领域，这类搜索通过更为深入地挖掘文本来定义相关性。这种专家搜索依赖于对专业人员（如律师或医生）输入的行业术语的理解。这类解决方案必须掌握专业领域里各种事物之间的微妙关联——比如，“Heart Attack”和“Myocardial Infarction”是一回事；或者急性“Myocardial Infarction”是一种特殊类型的“Heart Attack”。

正如电商搜索反映了顾客与销售之间的交互一样，专家搜索就好比是搜索者与研究室管理员之间的对话。这些管理员通晓专业研究人员的术语。当被问及一个问题时，他们会指导专业人员，找到那些专业人员自己不大容易找到的数据和相关研究成果。

有关这类搜索应用的相关性，其基本定义来源于为图书馆组织信息而建立的解决方案。例如，在医学上，如图 1.2 所示的 Medical Subject Headings（医学主题词表，简称 MeSH）分类法，对医学概念进行了有效组织，以帮助人们获得有关同义或近

---

<sup>1</sup> 详情请参见 Sergey Brin 与 Lawrence Page 共同撰写的 *The Anatomy of a Large-Scale Hypertextual Web Search Engine* 一文，位于 <http://infolab.stanford.edu/~backrub/google.html>。

义主题词的信息。对专家搜索而言，相关性意味着，在搜索查询与被搜索内容之间，将主题词与所要讨论的话题审慎关联起来。一个具备相关性的搜索结果就像是给被难住的研究人员带来的“灵光乍现”（“Aha!” moment）——一种他们自己不太容易找到的“顿悟”。

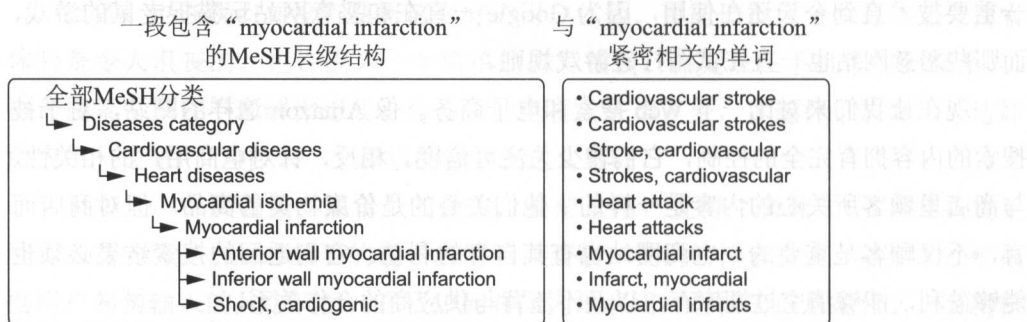


图 1.2 “myocardial infarction” 的 MeSH 分类（左侧）以及某些与 “myocardial infarction” 紧密相关的 MeSH 主题。

## 1.2.2 搜索：没有银弹

我们前面所讨论的几类搜索问题，不过是令人眼花缭乱的各种搜索应用的冰山一角。房产搜索是一种电商搜索吗？两者固然有相似之处（让用户买得称心），但对于一名购房者而言，还有许多其他因素需要考虑（优质学校、邻里关系、卧室数量）。本地餐饮的搜索应用呢？日用百货的搜索呢？用餐馆菜单来订餐呢？志愿者活动的搜索呢？或者是暴风雪过后寻找给马路铲雪的工作人员呢？企业内网的搜索呢？还有你自己的应用呢？你是怎样定义相关性的呢？

相关性需求如此多样，竟然还有这么多厂商热衷于提供“包治百病”的银弹式解决方案，真是不可思议。应用的相关性定义也许比你想象的还要独特。用户甚至有可能还没有意识到自己的需求是什么。而你还没有觉察到内容和业务带来的挑战。

真的要感谢 Solr 或 Elasticsearch，它们在不加改造的情况下是无法解决问题。我们没有从选择编程语言做起，因为我们的产品不过是一个从 Solr 或 Elasticsearch 的标准库衍生而来的模块。如果是这样的话，我们的产品就没有自己的特色了！相反，让我们把 Solr 或 Elasticsearch 想象成一种搜索编程框架。一个开源的搜索引擎允许你通过编程的方式将你对相关性的理解植入搜索引擎。我们将告诉你：利用开源搜索技术打造相关性解决方案的方方面面，既满足用户需求，又符合业务目标。



## 1.3 来自相关性研究的启示

好了，如你所见，我们的应用有它自己对相关性的定义。但是为什么没有一种一劳永逸的现成实践，能将相关性搜索结果呈现给用户呢？上网一搜，我们就会发现有很多一次性的解决方案，很好地解决了提出的各种问题。由此给人造成了一种印象，搜索的相关性并没有系统性的基础和通用的工程性原则，有的只不过是一堆无法广泛使用的奇技淫巧。

事实上，在相关性的背后的确藏着一门学问：学术领域里的信息检索（information retrieval）。它包含被普遍接受的各种实践，可以对广泛分布在各行各业的相关性加以改进。但是你也看到了，相关性是什么在很大程度上取决于你的应用。鉴于此，在我们介绍信息检索的时候，不妨思考一下，如何将信息检索的那些通用理论用于特定领域里相关性问题的解决。<sup>1</sup>

### 1.3.1 信息检索

幸好，专家们研究搜索技术已经好多年了。信息检索在学术领域关注的是准确返回信息，满足用户的信息需求（information need）。什么是信息需求呢？不妨把它看作一种规格要求，描述与用户搜索相匹配的理想内容。这种规格要求不仅限于查询字符串。例如，设想你正在试图解决一个编程问题。也许你在试图搞明白为什么Java库函数 `sort` 抛了一个 `NullPointerException`。此处的信息需求可以被描述为：

一个关于我在调用 `sort` 方法时遇到 `NullPointerException` 的解决方案。（尽管不好意思承认，但是如果有那么一段代码能让我复制粘贴了事，然后去吃午饭，那就太好了！）

为了满足这一信息需求，我们可能需要构造一些查询条件，以寻找解决这一问题的办法——例如，“`sort` 方法 `NullPointerException`”或“<代码片段> `NullPointerException`”。幸运的话，我们会搜到一则解决类似问题的信息。这一信息可以帮助我们解决问题，然后去吃饭。

在信息检索中，相关性被定义为一种返回搜索结果的实践，它能最大限度地满

---

<sup>1</sup> 有关信息检索领域的介绍，我们强烈推荐由Christopher D. Manning等人撰写的经典著作*Introduction to Information Retrieval*（剑桥大学出版社，2008），详见<http://nlp.stanford.edu/IR-book/>。





研究人员利用判定列表的目的——为了针对每一组测试文档的搜索结果，衡量对文本相关性算法的修改是否在整体上提升了相关性。对典型的信息检索来说，如果一个解决方案对若干包含大量文本的测试集在整体上能将相关性提升 1%，那就是一个成功的解决方案。信息检索关注的是如何解决一组广泛意义上的搜索问题，而不是深入到某个具体的问题中去。

### 1.3.2 能否利用信息检索解决相关性问题

我们已经看到了世上没有银弹。但是信息检索看起来的确系统化地给出了一些相关性的解决方案。因此，问问你自己：这些方案适合你的应用吗？你的应用是否关注这样的解决方案，它为文章搜索提供了渐进式的还是一般性的改进？如果用针对性的方法来解决应用此时此刻所面临的问题，是不是会更好呢？

更确切地说，经典的信息检索技术在被用于解决相关性问题时，有一些问题是被回避了的。让我们来仔细考察一下这些问题，看看信息检索在哪些方面能够帮到我们，哪些方面则爱莫能助。

- 我们是否只关心信息需求呢？对很多应用而言，满足用户的信息需求并不是唯一的目的。搜索的存在也要满足应用背后的业务需求。这一点我们在此前的电子商务中已经看到了。尽管人们常说“客户至上”，但是如果不卖广告，不创造利润，不满足供应商，不让库存流动起来，企业也是无法运转的。任何搜索体验中都存在着各种各样的因素，可以令业务需求凌驾于用户的信息需求之上。就好比二手车卖家总想把旧车高价出售一样，相关性技术的研发工程师必须妥善处置这些因素，让雇主能够稳赚不赔。
- 除了文本还有什么可以反映信息需求呢？经典的信息检索技术关注的是一种通用的、一劳永逸的文本相关性度量方法。它所依赖的那些因素也许对你的应用而言根本无关紧要。我们需要更多地关注待处理问题的特殊性。前面我们已经讨论过一个例子：Google 通过引入一种数字化的网站信用度量方法（PageRank），彻底改变了 Web 搜索。凭借 PageRank，Google 避免了使用纯基于文本的度量方法，在它的领域里游刃有余。即便是文本搜索，也不总是像信息检索那样，只关注大段的文字。要想从 tweets 或 titles 这样的短文本中得到满意的搜索结果，需要不同的思路。对此，你（而非信息检索研究人员）必须自己来决定哪些因素对应用而言是有价值的，然后将其实现。一种

针对 Reuters 测试集表现不彰的方法，也许恰恰是你需要的，因为它满足了你的用户需求。

- 用户的体验蕴含了什么样的信息需求呢？通常应用自身所承诺的功能就决定了用户对相关性的理解。此前我们曾经讨论过专家搜索。试想两个医疗搜索应用，均服务于同一类人群（医生），掌握着相同的数据（医学论文）。但是有一点重要的不同：一个应用帮助医生服务于临床病患，而另一个则允许医生偶尔在办公室搞搞研究。这两种对系统截然不同的期望，就意味着对同一搜索查询，其相关性的不同理解。在病房里搜索“heart attack”必须给出行之有效的、安全可靠的解决方案，那是人命关天的问题。而那个科研性质的应用则允许更为多样的查询结果：医生可以通过搜索“heart attack”来探寻有趣的最新研究发现，这些发现并不局限于解决具体问题。

通常成为一名相关性技术工程师最难的是要理解上下文与信息需求之间的关系。送达搜索引擎的用户查询附带了大量的信息。这些信息有一部分是以附加数据的形式出现的，如地理位置或用户会话。但还有一些信息则是完全隐含在搜索应用所承诺的功能里的。应用的开发、销售和推广，其目的是为了放在某人的案头偶尔搞点小研究呢？还是几乎被当成一个专家系统来售卖，时刻准备着，愿意且有能力强解决用户提出的任何问题，包括帮助医生挽救生命？

考虑一下前面提到的这些问题，可以看到，信息检索技术为我们建立起了一个基础，能够将一些普遍适用的相关性度量方法广泛应用于各类问题。而你的工作，则是要解决自己应用的相关性。正如我们看到的那样，这些问题很大程度上游离于搜索技术之外，且涉及更为广泛的各种产品策略问题：我们的用户是谁？他们希望通过这个应用得到什么？搜索要解决的那些隐含在其中而未加指明的信息需求是什么？

在继续往下讨论之前，让我们来完善一下对相关性的定义，它是采用什么样的手段来解决实际的相关性问题的：

相关性是一种改进用户搜索结果的实践，它在用户体验的具体上下文中满足用户的信息需求，同时平衡排名对业务需求的影响。

## 1.4 如何解决相关性

了解了信息检索，让我们把注意力转向如何解决我们自己的相关性问题上。一些开源的搜索引擎认识到应用的相关性取决于各种因素。这其中有很多因素都是和具体应用息息相关的（比如，用户距离餐厅有多远）。而另一些因素则更具有广泛性和普遍性，它们所涉及的就是信息检索中的文本排名。

有了开源搜索引擎，我们如何解决实际的相关性问题呢？我们能定义出什么样的框架来，既包含狭义的领域特定因素，又用到广义的信息检索技术呢？

为了解决相关性，工程师需要：

- 1 识别出能够刻画内容、用户或搜索查询的关键特征。
- 2 通过对特征的提取和对内容的丰富，想办法让搜索引擎理解这些特征。
- 3 在搜索期间，通过构造信号来对用户搜索的相关性加以度量。
- 4 在对结果进行排名时，通过控制排名函数，仔细平衡多个信号之间的影响。

这一流程如图 1.4 所示。

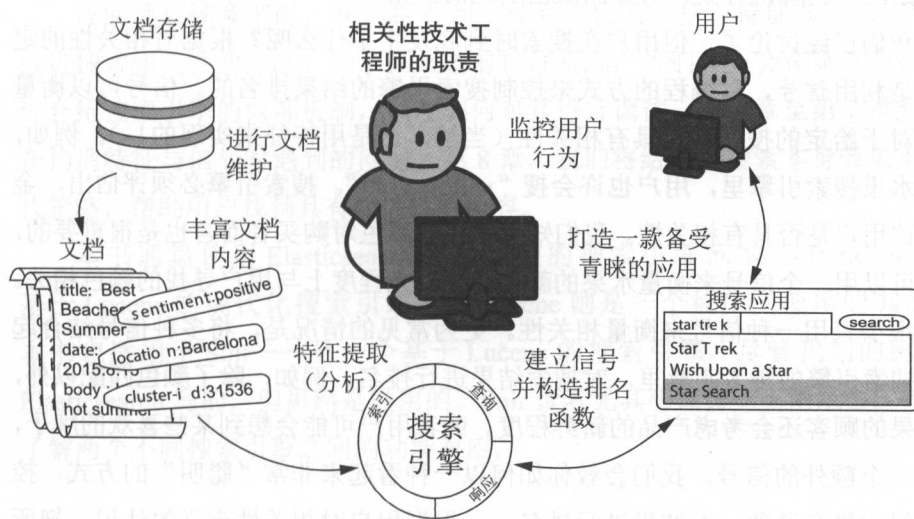


图 1.4 相关性技术工程师从后端系统中选取、补充或构造重要特征，并定义有关这些特征的排名信号。

这看起来有点抽象，到底是什么意思呢？前面我们讨论过一个例子：Google 为网站找到 PageRank 这一特征（步骤 1）。这一特征伴随着每一个网页被“编”进了

Google 的搜索引擎里（从而实现了步骤 2）。当你发起一次搜索，Google 会根据你在这次搜索中所考虑的相关性对各种因素进行度量（步骤 3）。例如，Google 会直接将 PageRank 作为一种可信度的排名信号。其他信号可能还包括：搜索的字符串在页面标题或正文里出现的频率，以及基于用户偏好的个性化因素。Google 将所有这些信号都放入一个更大的排名计算程序中，对搜索结果进行排序，以期让用户找到满意的结果（步骤 4）。

这些观点已经在本章前面部分讨论过了。但在这里我们还是要给出一些更为准确的定义。所谓特征，是内容（content）或查询（query）的一个属性。特征影响决策。搜索相关性的研发工作很大程度上就是在做特征选取——一种发现和构造特征的行为，让用户在搜索时为我们提供相应的信息。

熟悉机器学习或分类学的读者也许能找出一些可识别的特征来。当你在做分类时，会从数据中寻找新特征，做出更好的分类决策。这个水果是香蕉还是苹果呢？如果你知道它的颜色是黄色的，那么它可能就是香蕉。如果我们增加一个关于形状的特征——圆的还是长的——那就能做出更明确的判断了。你会发现，这些特征也会帮助搜索应用的解决方案，对数据做出明确的决策。

特征我们已经讨论了，但用户在搜索时到底发生了什么呢？根据对相关性的定义，我们是利用信号，以编程的方式来控制搜索引擎的结果排名的。信号可以衡量一样东西对于给定的搜索是否具有相关性（当然，这是用特征来实现的！）。例如，在我们的水果搜索引擎里，用户也许会搜“yellow fruit”。搜索引擎必须评估出，金冠苹果对该用户是否具有相关性。我们知道水果的颜色对购买者而言也是很重要的，因此也许可以用一个信号来衡量水果的颜色，在多大程度上与用户寻找的颜色相关。

人们很少只用一种信号来衡量相关性。更为常见的情况是，将多种信号结合起来，放到搜索引擎的排名函数里，对搜索结果进行排名。例如，除了颜色匹配以外，也许买水果的顾客还会考虑产品的新鲜程度。或者用户可能会想到某些喜欢的牌子，将其作为一个额外的信号。我们会教你如何以一种看起来非常“聪明”的方式，控制搜索引擎的排名函数，对结果进行排名——根据用户对相关性定义的认识，把所有要考虑的因素（即信号）都计算在内。

别担心——我们知道到目前为止这些想法都还有点抽象。随着在后面章节里的实际动手操作，到时候你就会瞬间明白我们在这里所说的了。不过为了能对特征有一个大概的认识，让我们来看几个特征的例子，以及它们是如何被用于排名时的搜

索信号的。

- 销售数据，用户评分——这些特征用来反映结果的受欢迎程度，东西越受欢迎，用户也许就越开心。
- 带有位置信息的文本——用来反映用户查询条件中出现的词汇与搜索内容的匹配情况。
- 带同义词的文本——查询条件中出现的同义词是否与搜索内容相匹配。
- 地理位置——某样东西的距离是远是近：用户和搜索内容离得近吗？寿司店就在附近，还是在曼哈顿？
- 机器学习/分类学特征——用户的搜索是否更容易被划分到某一类内容中去（如对电影的搜索），而不太容易被划分到其他类型的内容中去（如对除草机的搜索）？
- 个性化/推荐——用户是否表现出对某种特殊类型的内容比其他内容更感兴趣？你能识别出与该用户兴趣相仿的其他用户吗？也许发起搜索的用户，其偏好历史也可以被用作信号，影响搜索的结果。

在阅读后续章节时，我们将看到一种系统化的、基于特征选取和信号编程的、改进搜索相关性的方法。为了在此之前打好基础，我们首先会在第2章和第3章向你介绍搜索引擎的内部机制，以及如何对其进行调试。第4章至第7章介绍了许多在构造特征与信号时遇到的问题。第8章，我们将给出在搜索本身效果不佳时的替代策略，帮助用户找到具有相关性的内容。

本书通篇使用 Elasticsearch 作为我们的搜索引擎范本。Elasticsearch 是一个基于 Lucene 的现代化搜索引擎，而 Lucene 则是一个被广泛使用的 Java 搜索库。本书也适用于 Solr——另一个基于 Lucene 的搜索引擎。尽管我们的例子是基于 Elasticsearch 写的，但思路是通用的。Solr 读者尤其应该读一下附录 B，有助于大家了解两个不同搜索引擎之间的功能对应。

## 1.5 不只是技术：管理、协作与反馈

以技术为后盾是否足以解决搜索的相关性问题呢？掌握着从本书学到的新技能，也许你跃跃欲试地想要改进雇主的搜索系统。你对目标的定位是最大的相关性问题，你交付给用户自认为最棒的搜索体验。不费吹灰之力就发布了几个更新版本；



对公司而言，这不过又是众多埋头工作的一幕，工程师们遇到问题，然后轻松搞定。这有点像为 SQL 数据库提升性能一样，不是吗？

不幸的是，系统发布之后不久，老板就找上门来了。情况看起来很糟糕。尽管你使出浑身解数，系统还是出了严重的问题。不知怎么的，用户就是无法购买商品。他们找不到他们想要的信息。相反，他们正在选择放弃并转向竞争对手的系统。随着营收的下降，老板牙关紧锁。绝望中，她看着你恳切地说“一定要提高相关性啊！”换句话说就是，修改问题，开发功能——不行的话就周末加班；不过是为了让系统工作正常！

“一定要提高相关性”？让我们回忆一下我们对相关性的定义。如果你仔细回想一下定义，也许就会发现故事里的这家公司恰恰忽略了这一点。

相关性是一种改进用户搜索结果的实践，它在用户体验的具体上下文中满足用户的信息需求，同时平衡排名对业务需求的影响。

当你思考这个定义时，很快就发现工程师们并不清楚相关性搜索应该长什么样！为了满足用户的信息需求，你需要理解他们的目标、领域，以及他们搜索的上下文。这些情况可能千变万化，从医生救死扶伤，到老人给孙子降生买见面礼。满足这些用户的需求，就意味着要钻入他们的大脑。急用户之所急，远远超出了搜索技术的范畴，它几乎要牵涉企业里方方面面的人。这一点在你理解业务需求的过程中尤其如此，比如政策、营收、业务目标，还有其他内部因素。

解决搜索的相关性问题要求人们将企业文化转换到跨职能协作上来。企业如何教会相关性技术的研发工程师理解用户的语言，以及用户希望通过搜索得到什么？如果应用是服务于医生或律师的，情况又会怎样？谁来帮助工程师理解这些用户所处的行业？企业如何让工程师了解在公司里什么业务最赚钱？哪些供应商不能得罪？搜索中哪些内容要有“付费”入口（甚至这到底是什么意思）？

即使看似最普通的搜索应用，也可能充满了这样错综复杂的情况。比如一个餐饮搜索应用。市场部的同事费了九牛二虎之力把用户“领进了门”。现在，搜索应用将扮演起网站销售的角色（或者应该叫“门房”），它需要满足用户的需求，并能够让他们乐于当回头客。

但是，相关性技术研发工程师毕竟不是来自销售部门。当一个用户在搜索框里输入“sushi”时，这个用户期望看到的是什么样的餐馆呢？普通外卖？高档餐厅？

临近商铺？还是用户自己决定呢？企业中的另一些人，他们不是技术人员，却懂得用户希望获得的目标是什么。工程师在孤立的环境下定义相关性排名，就是在闭门造车。

更进一步说，这种技术人员与业务人员的协作，超越了单纯意义上对工程师的培训。管理，对内容的管理，以期它能很容易地被用户搜到，也许和对工程师的培训是同等重要的。回想一下在本章前面讨论过的那个专家搜索的例子。在那个例子里，管理员的专业知识能够帮助我们有效地组织内容，让其更易于被找到，从而帮助你构建更好的搜索应用。通常，企业必须把那些对内容有深入理解的业务人员，和那些熟悉搜索引擎工作机制的工程师，叫到一起坐下来开会才行。

这种协作形式背后体现的是反馈的理念。一家高效率的企业会致力于相关性技术的研发工程师带来准确快速的反馈，以此来为之提供信息并指导其工作。如图1.5所示，我们可以直观地看到几个重要的反馈环，它们以一系列关注度递增的圆环形式呈现出来。从最外层的循环开始，企业里搜索应用的开发人员悠然自得，对搜索相关性的影响不甚了了。伴随企业的发展，企业向内侧、更为成熟的反馈形式移动：考虑到了用户行为数据和专家反馈。最后，企业将它的睿智编入相关性测试的代码里，成功推动了测试驱动的相关性实践——这是最成熟的组织形式。



图 1.5 搜索相关性的反馈形式

本书主要向你讲述的是相关性技术的研发工程师所要掌握的技术。但是在学习



这些技术时，希望你能不断地在脑海里提醒自己，我们应该做什么。在本书的很多例子里，我们会反复强调，用户想看到的是特定的搜索结果。我们以这种方式向你传授研发搜索应用的技艺，以得到用户想要的结果。在你学习这些例子时，请记住本节的例子，当你急于想将本书中学到的知识用于解决实际问题时，先想想这些例子。在第10章，我们还将深入讨论企业里的种种挑战。

## 1.6 本章小结

- 相关性问题无处不在。即使是那些成熟的领域，如 Web 搜索、电子商务，还有专家搜索，人们依然在努力提高搜索结果的相关性。
- 将用户导向相关性搜索结果可以带来数十亿美元的生意；反之则意味着将蛋糕拱手让给竞争对手。
- 为用户带来满足其信息需求的内容，在学术领域被称为信息检索。这里的信息需求主要是在搜索的查询条件中定义的。
- 在实践中，相关性不仅意味着满足由查询条件所指定的信息需求，它还意味着满足业务需求。而且，理解用户的信息需求常常要依赖于一些隐含信息，比如应用的上下文、目标、市场，以及用户体验。
- 要实现相关性，需要从应用的上下文中识别出有价值的特征，还要利用这些特征计算出相关性信号。
- 技术人员无法独自完成工作。取决于业务需求、用户受众，以及内容涉及的领域，相关性技术工程师往往欠缺相应的技能来评估什么内容对用户搜索具有相关性。
- 反馈是很重要的。从工程师的角度来看，对相关性的改变所带来的影响加以度量，有助于避免为用户提供劣质的搜索结果。

# 搜索——幕后揭秘

## 本章要点

- 理解搜索技术所需的基本概念
- 令搜索得以完成的数据结构
- 文档搜索和检索的内部机制
- 有关数据的提取、充实、分析和索引的概述

搜索是用户和搜索引擎之间的一种人机对话。用户通过为搜索引擎提供描述相关性内容的合适的查询条件，以尝试满足其信息需求。搜索引擎利用这些查询条件来搜索匹配项，并将之返回给用户。如果用户满意匹配结果，就会进一步查看每一个返回项的具体内容。否则，用户就会完善查询条件尝试再次进行搜索。我们的工作就是辅助这场人机对话，务必确保搜索结果的相关性，并帮助用户理解为什么结果是相关的，使他们能够得到更满意的搜索体验。

然而实现相关性搜索所需要的，远远出乎我们的意料。初来乍到的搜索工程师通常认为搜索就是一个黑盒，只包含少数几种交互模式：将内容加入搜索引擎，然后允许用户对内容进行查询。本章我们将揭开它的神秘面纱。在这里剧透一下，搜索其实也没什么特别神奇的地方。在底层，一个搜索引擎的数据结构执行着相当机

械的词匹配 (term-matching) 操作, 然后对结果用简单的启发式方法加以排名。搜索引擎是机械的: 它不会理解词 (term) 的含义、用户搜索的潜在意图, 以及应用的上下文背景。

相关性工程 (relevance engineering) 是一门运用机械化的搜索引擎来实现相关性目标的学科。在本章结束之前, 大家将会看到我们是如何做到这一点的。我们将学会如何从查询和内容中提取描述性的特征, 理解这些特征如何被用来对文档进行排名, 将最相关的文档排在前面。除了相关性工程学, 我们还会懂得良好的、具有描述性的特征是由什么构成的。再就是如何加权与平衡各种因素, 以同时满足用户和企业二者对相关性的诉求。看完本书, 大家就能很好地掌握这些技术了。

在本章中, 我们首先简单介绍基本的搜索概念, 然后基于 Lucene 深入讲述一些搜索技术的细节, 包括数据结构和文档的分析、索引以及检索的流程。第3章, 我们将会看到在对搜索进行分解时所要去做的工作。第2章和第3章为帮助读者了解搜索引擎提供了基础性的工具。接下来的章节将进一步讲述如何应用搜索的数据结构来解决现实生活中的问题。

## 2.1 搜索101

我们对搜索引擎最初的理解也许是简单的。凭借所掌握的一些基本知识, 在我们的头脑里大致可能会有一个如图2.1所示的模型。内容进入搜索引擎, 用户通过与搜索应用的交互来查询和检索想要的内容。



图 2.1 基于可能存在的交互, 一个简单的搜索引擎模型。

在深入探究搜索这个神秘的黑盒之前, 让我们先从外部来快速回顾一下搜索引擎的功能。众所周知, 搜索引擎的核心功能是存储、查询并获取内容。尽管这些都是基本概念, 但为了确保在概念理解上形成共识, 并对技术查漏补缺, 将它们重新梳理一遍总是没有坏处的。

### 2.1.1 什么是搜索文档

在搜索应用中，文档（document）这个概念是核心，因为存储、搜索和返回的都是文档。文档是搜索的全部！当我们向搜索引擎发起一次查询请求时，实际上是在搜索文档的一个集合。这些文档可以是实实在在的文档，比如保存在服务器上的文本文件，或者更加广义一点，文档也可以是如下这些内容：

- 商品目录中的商品
- 存储在 MP3 播放器中的歌曲
- 联系人列表中的联系人
- 来自公司内网的内部 Word 文档
- 一本书的书页
- 图书馆馆藏的全部书籍

一篇文档包含了一组字段（field）：即文档的具名属性。这样一来，文档就有些类似于 SQL 表中的行了。只不过 SQL 表中包含的是一组具名的列及其对应的列值，而文档所包含的是一组字段以及字段值。字段是有类型的，包括我们所熟识的那些标准类型：字符型（string）、整型（integer）、浮点型（float）和布尔型（Boolean）。

字符串类型尤其受搜索的青睐。人们经常在字符串中进行搜索。考虑一篇以“Business Is Booming at the Beautiful Beach of Barcelona”为题的新闻，我们也许会通过搜索字符串“barcelona beaches”来匹配标题中的文本。作为相关性研发工程师，如何准确控制文本来匹配搜索会花费掉我们大量的时间。这一点目前你可能还有点不太理解，但是看完本书，尤其是第 4 章，你就明白了。

与 SQL 表不同，每篇文档可以包含不同的字段。一篇文档中的字段可以和另一篇文档中的字段不尽相同。假设我们正在为连锁便利店的商品构建搜索，便利店里各种各样的商品。尽管所有商品都有一些公共字段（例如名字、价格），但是绝大多数商品仍需要它们自己特有的字段。在便利店里，书籍需要诸如“作者姓名”和“页数”这样的字段。食品可能需要诸如“卡路里”和“原料”这样的字段。

### 2.1.2 对内容进行搜索

有了文档的概念，我们就可以讨论搜索引擎的主要目的了：搜索相关性内容！假设你的朋友 Sharon 想分享给你一篇关于巴塞罗那海滩的非常棒的文章，她记得这

篇文章发表于去年夏天《Relevant 时报》(Relevant Times)的旅游(Travel)版面。为了将那篇文章分享给你,她打开 RelevantTimes.com 的搜索页面,如图 2.2 所示,并搜索满足下列查询条件的文章:

- 发表于 6 月、7 月或 8 月(以日期范围为限定条件)。
- 在旅游版面(字符型字段的精确匹配)。
- 关键字 barcelona 和 beach 出现于标题和正文字段中(在两个文本型字段中的文本匹配)。

The screenshot shows the Relevant Times website's search interface. On the left, there's a search bar with 'barcelona beaches' entered and a 'search' button. Below the search bar, there are date filters for 'June 1, 2015' and 'Aug 31, 2015'. To the right of the search bar, a table lists search results with checkboxes for different categories: News (1 result), Political (0 results), Travel (12 results, checked), Economy (5 results, checked), Lifestyle (0 results), and Opinion (1 result). On the right side of the interface, there are three article snippets. The first is 'Business is Booming on the Beautiful Beaches of Barcelona' by Ted Nisemono, dated June 23, 2015, in the Economy section. The second is 'Best Beaches to Beat the Summer Heat' by John Faux, dated Aug 14, 2015, in the Travel section. The third is 'Barcelona Festival Season is Upon Us' by Cindy Falshung, dated July 3, 2015, in the Travel section. Below these, there's another snippet 'Stinging Jelly Fish Plague Shoreline Industries' by Phil Lazan, dated June 3, 2015, in the Economy section.

**Relevant Times**  
Your News... Yesterday!

barcelona beaches search

June 1, 2015 ▾ Aug 31, 2015 ▾

no. results

<input type="checkbox"/> News	1
<input type="checkbox"/> Political	0
<input checked="" type="checkbox"/> Travel	12
<input checked="" type="checkbox"/> Economy	5
<input type="checkbox"/> Lifestyle	0
<input type="checkbox"/> Opinion	1

**Business is Booming on the Beautiful Beaches of Barcelona**  
Ted Nisemono Economy June 23, 2015  
... this summer season is projected to be the best on the books for **Barcelona's beach** resorts, a trend that is expected to continue ...

**Best Beaches to Beat the Summer Heat**  
John Faux Travel Aug 14, 2015  
... there are several **beach** destinations that you must visit ... or Waikiki which as placed just above **Barcelona** as a destination ...

**Barcelona Festival Season is Upon Us**  
Cindy Falshung Travel July 3, 2015  
... whether it's the giant's on the march or the 12 story tall human pyramids you'll find **Barcelona** an enchanting place to spend ...

**Stinging Jelly Fish Plague Shoreline Industries**  
Phil Lazan Economy June 3, 2015  
... the northside beach was worst hit, but will a little help from a ...

图 2.2 典型的用户搜索和返回界面

一旦提供了这些限定条件,搜索引擎就会返回匹配的文档。不仅如此,搜索引擎还会对文档进行排序,将最相关的匹配项优先提供给用户。

在 Sharon 的“barcelona beaches”搜索中,搜索引擎有望能将其正在寻找的文章排在第一位。搜索引擎是如何知道要给这篇文章较高排名的呢?那是因为这篇文章是有关巴塞罗那海滩的,文章的标题里可能包含短语“Beaches of Barcelona”,单词“Barcelona”和“beach”可能也会显著地出现在文章的内容中。搜索引擎综合考虑了这些因素,并将 Sharon 要寻找的目标文章显著地排在了其他文章的前面:即那些与用户搜索的相关性较低但仍然保有一定相关性的文章。或许这些文章是关于“beaches”或“Barcelona”的,但本身与“the beaches of Barcelona”并无直接关系,只是在搜索查询中稍有提及罢了。

稍后我们将会看到更多关于搜索引擎如何准确进行相关性计算的内容。我们还将看到,除了文本、用户的优先级,还有哪些因素可以在相关性计算中发挥作用。在第 1 章中我们提到过,相关性是衡量搜索结果在多大程度上满足用户和业务的信

息需求的一种手段。比如，一个具备良好相关性的电商搜索不仅要返回合适的文档，还要保证或尽可能保证这些文档被恰当排序，让促销商品排在更加靠前的位置。

### 2.1.3 通过搜索来探索内容

搜索引擎的作用不仅仅局限于返回相关性文档。如果不能以一种鼓励用户继续进行探索的方式将相关性文档呈现给用户，那么搜索的作用将大打折扣。利用大家已经耳熟能详的许多常见的功能特性，搜索引擎的用户界面可以引导用户获得相关性的内容。

在如图 2.2 所示的最显著区域，搜索引擎为用户提供了一个匹配文档的列表。通常用户不会去浏览整篇文档，而是只关注全部字段的一个子集，这几个字段对于理解文档匹配与否至关重要。在“巴塞罗那海滩”的例子中，这些字段可以是文章的标题、作者、报纸的版面（本例中是旅游版面）、日期，以及取自文章本身的文字内容。

搜索引擎通常并不返回字段的所有内容，而是返回一些总结性的片段，并高亮显示匹配部分。这些被高亮显示的片段（简称为高亮）准确地揭示了为什么这篇文档匹配用户的搜索条件。通过阅读这些片段，用户通常会找到办法改进自己最初的搜索条件。例如，如果“Barcelona Beach”碰巧是当地一家小饭馆的名字，那么用户很可能会修改之前的查询条件对饭馆进行过滤。

搜索引擎还会通过描述所匹配的文档在整个文档集中的分布情况，来鼓励用户进一步优化搜索查询条件。例如，Sharon 对巴塞罗那海滩的搜索原本只局限于旅游版面的文章。但是一个好的搜索引擎也许会显示出报纸其他版面中匹配“barcelona beaches”的文章数量。如图 2.2 所示，这样的聚合信息经常被当作一组过滤条件（也被称为切面，英文为 facet）出现在侧边栏中。或许 Sharon 记得读过的那篇文章是有关巴塞罗那海滩蓬勃发展的旅游业的，它出现在报纸的经济（Economy）版。那么根据切面中给出的数据，Sharon 也许会选择该切面，对搜索进行过滤，只包含来自经济版的文章。

### 2.1.4 获取进入搜索引擎的内容

前一节中我们讨论了如何通过搜索优化查询条件，但是如何为搜索引擎提供这些可以被搜索到的内容呢？首先，我们从内容存储的地方提取数据，这个地方可以



是数据库、文本文件、网页，或其他来源；然后，将原始数据转化成前面所讲的搜索字段和文档。通过加入一些包含外部信息的、有助于匹配与排名的新字段，这些文档可能会得到进一步充实。

等进入搜索引擎之后，文档的字段会经历一个被称为分析（analysis）的过程，如图 2.3 所示。



图 2.3 被索引和分析过的“Barcelona Beaches”文章（图中仅包含了标题字段的分析）

分析将字段值（通常是文本）转换为被称为 token 的元素。对于文本而言，token 通常对应于单词，例如“best”“barcelona”“beach”。读者可能注意到了，这些 token 看起来和初始的单词都有点不太一样。在本例中，单词“the”被去掉了，所有 token 都变成了小写，“beaches”的复数后缀也被去掉了。为什么要这样做呢？这是因为只有提取自文章中的 token 和提取自查询中的 token 达到精确匹配时，才能被认为是匹配成功的。搜索经常借助特定语言中的启发式处理（heuristics）将单词转换成简单格式来帮助匹配。英语的文本分析会去掉大写（RUN → run）、后缀（running → run）和复数（runs → run），以及许多其他的格式。并且尽管一般来说 token 都是文本的，不过在第 4 章中，我们将会看到一些通过分析得到的非文本的 token，例如浮点数和地理位置。

在第 1 章中，我们提到了特征的概念。在机器学习中，特征是对内容进行分类时，描述内容的手段。例如像颜色、味道和形状这样的特征常被用来对水果进行分类。在全文检索中，分析过程中所产生的 token，是在将用户查询和索引中的文档进行匹配时，所采用的主要特征。如果现在对这些还有点不太明白，请不必担心，本书后面将会有很大一部分篇幅专门深入阐述这些内容。

完成分析之后，文档就会被索引；从分析环节得到的 token 会被存储到搜索引擎的数据结构中，用于文档的检索。此外，最初的、未经分词处理的文本字段也会



被保存下来，以便将其显示在查询结果中返回给用户。那些数值型字段也会被保存，以允许我们在排名计算中使用数值属性。

## 2.2 搜索引擎的数据结构

2.1 节介绍了“搜索入门”的基础知识。现在到了本章最为精彩的部分了！我们将开始填补那个充满魔力的黑盒，理解搜索引擎的中枢系统是如何运转的。我们将从数据结构开始，然后再讲述在索引（indexing）和查询（querying）这两个过程中如何运用这些数据结构。

搜索引擎的核心部分是为数不多的几个支持文档检索和评价的、高度优化的数据结构。理解这些数据结构以及如何使用这些数据结构，是理解搜索引擎内部机制的先决条件。基于对这些机制的理解，我们就可以使用搜索引擎来营建看似智能的相关性搜索体验了。

### 2.2.1 倒排索引

我们希望，将来当有任何相关性搜索的问题困扰大家时，大家能想到再次翻开这本书来寻求帮助。如果是这样的话，我们该从哪里着手呢？十有八九我们会直接翻到本书的最后，查找索引（参见图 2.4 所示）。<sup>1</sup> 在那里我们能够查到自己感兴趣的术语：analysis、tokenization、scoring 等。然后据此找到对应的书页，那里有我们想要了解的内容。

<b>A</b> acronyms 90–91 actionable information 186 ad hoc searches 148 add shingling 126 addCmd 45 additive boosting, with Boolean queries 176–178 combining boost and base query 177–178 function queries vs. 174–175 optimizing boosts in isolation 176–177 Solr 318 adjusted boosts 72 affinity 286 aggregate information 20	AND operator 32–33 anticipating user behavior 93 user intent 76 api_key argument 304 assertion-based testing 273–274 asymmetric analysis 96, 100 asymmetric tokenization 101 autocomplete keyword 255  <b>B</b> bag of words model 63, 65 base query, boosting 173 base signal 176 basic highlighter 225
---	--

图 2.4 搜索引擎使用的倒排索引数据结构非常类似于书籍后面的索引

<sup>1</sup> 由于本书索引中名词术语对应的页数是英文原书的页码，在中文译本中不具参考意义，因此并没有出现在中文译本中，在这里只是举一个例子进行说明。——译者注

搜索引擎的核心部分是一种被称为倒排索引 (inverted index) 的数据结构, 它类似于图书后面的索引。倒排索引由两个主要部分组成: 词典 (term dictionary) 和倒排表 (posting list)。词典是给定字段在一个文档集中出现的所有词汇所组成的有序列表。词典中的每个词都有一个包含该词的文档列表与之对应。这个文档列表就被称为某一词的倒排表记录 (posting)。为了更清楚地理解这一点, 让我们看一个例子。请看清单 2.1 所示的一组文档。

### 清单 2.1 文档

0. One shoe, two shoe, the red shoe, the blue shoe.
1. The blue dress shoe is the best shoe.
2. The best dress is the one red dress.

这个简单的文档集合, 其词典和倒排表分别如清单 2.2 和清单 2.3 所示。

### 清单 2.2 词典

best	→ 0
blue	→ 1
dress	→ 2
is	→ 3
one	→ 4
red	→ 5
shoe	→ 6
the	→ 7
two	→ 8

### 清单 2.3 倒排表

0	→ [1, 2]
1	→ [0, 1]
2	→ [1, 2]
3	→ [1, 2]
4	→ [0, 2]
5	→ [0, 2]
6	→ [0, 1]
7	→ [0, 1, 2]
8	→ [0]

词典和倒排表都是映射表。词典将词汇映射到一组能够唯一标识某个词的自然序数上。正如一本书的索引, 为了便于查找, 索引是按字母顺序排列的。一旦知道了一个词的序数, 我们就可以根据这个序数在倒排表中找到包含该词的所有文档——就像书籍索引中的页码一样。

为了更为清晰地理解这一点，让我们来看一个例子。假设我们要找所有包含单词“red”的文档。首先在词典里找到“red”，并且知道了它对应序数5。然后在倒排表中找到5对应的记录——本例中是文档0和2。参照清单2.1中的原始文档，我们可以看出，文档0和2确实包含“red”，而文档1则没有包含。

在本例中值得注意的是，我们对问题做了一定的简化。这些文档都只包含了一个字段，该字段包含了所有的句子。但在实际应用中，文档可能会包含多个字段：标题、描述、地址、价格等。即便如此，我们也什么都不用变；仍然只是一个倒排索引，只不过我们先将词汇按照字段分类，然后在字段内部则按字母顺序进行排序。

### 2.2.2 倒排索引的其他内容

词典和倒排表是倒排索引这种数据结构的核心部分，因为它们让文档快速匹配查询条件成为可能。但是为了让搜索引擎能够提供相关性结果并帮助用户改进查询条件，Lucene在索引中还加入了更多的数据结构和元数据。

这其中的多数组件都是可选的。有时候将它们禁用可能是一种优化方案，而有时候我们则可以启用这些数据结构，以获得更好的相关性或搜索特征。表2.1列出了通常与倒排索引相关的若干最为重要的信息。贯穿整本书，我们会进一步深入挖掘这些内容，并揭示如何利用它们来提高搜索的相关性。

表 2.1 与倒排索引相关的重要数据

名称	描述
文档频率	文档频率是指包含某个词的文档数量，或与某个词相关联的倒排表记录的长度。在清单2.3中，单词“shoe”的文档频率是2，因为它在文档0和1中都出现了。文档频率在文档评价中是有用的，因为它为某个词建立起了一种“重要性(importance)”的概念。例如，单词“the”通常有很高的文档频率，这表明它在决定一篇文档的相关性时具有较小的差异性
词频	词频是指一个词在某篇文档中出现的次数。在清单2.3中，“shoe”在文档0中的词频是4，在文档1中的词频是2。词频在文档评价中是有用的，因为它为一篇文档对某个词建立起了一种“重要性”的概念。因此，不严格地说，如果有人搜索“shoe”，我们认为文档0在重要性上可能要2倍于文档1，因为“shoe”在其中出现的次数是文档1的2倍
词位置	词位置对搜索而言往往十分重要。想一想查询“dress AND shoes”和查询“dress shoes”在语义上有何不同。词位置是一个数字列表，表明词在某篇文档中出现的位置。例如，本例中“shoe”在文档0中的词位置是1, 3, 6, 9。词位置使得基于短语匹配(phrase match)的文档查找成为可能，利用短语匹配搜索“dress shoes”会返回给用户准确的搜索结果

续表

名称	描述
词偏移量	这是为用户提供反馈，告知其为什么文档与搜索条件相匹配的最好方式之一，是为用户呈现经高亮显示的匹配文字的片段。但是通过对原始文本重新进行分析来提取高亮信息通常是很慢的。显示高亮的最快方法是在索引阶段首次对文本进行分析时记录下单词首尾字母的偏移量。然后，在搜索期间要做的全部工作就是将标记（tag）插入到对应的偏移量处
负载数据	索引中的每个词都可以与任意数据相关联。一个常见的例子就是给 token 加一个词性（part of speech）标记，用于相关性评价。另一个常见的例子则是给 token 加一个外部生成的评价价值（此处“Barcelona”的评价价值应为 100，另一个“Barcelona”则为 59）
存储字段	存储于倒排索引中的信息在搜索时是非常有用的，但是这些信息是原始文档的一个“混淆”（scrambled）版本。任何要返回并显示给用户的字段都必须单独保存于存储字段中。存储字段可能会占据很多磁盘空间。基于这一原因，许多搜索引擎的开发人员都避免直接将数据存储到搜索引擎内，而是从源系统中获取需要显示的字段
文档值	将一些辅助性的取值纳入相关性评价的启发式算法中是很常见的做法。例如，在电子商务搜索中，宣传一些清仓商品或者利润较高的商品是很常见的。允许用户按照某种度量对搜索结果进行排序也是很常见的，例如按价格或者销量排序。文档值这种数据结构允许我们快速访问这些辅助值，在对文档进行排序、评价和分类时非常有用

既然我们对支持搜索的各类数据结构有了较为清晰的理解，是时候来看一下信息是如何在一开始就被放入这些数据结构中的了。

## 2.3 对内容进行索引：提取、充实、分析和索引

2.1 节提到了文档如何进入索引的基本知识。在本节中，我们将深入探究这一过程，帮助大家理解文档是如何进入 2.2 节中介绍的搜索引擎的那些核心数据结构的。

将数据放入数据存储（data store）中需要经过信息的提取（extract）、转换（transform）和加载（load）过程，通常被缩写为 ETL。数据从数据仓库的各个地方被提取出来，转换成目标存储要求的格式，然后加载到目标存储中。在本节中，我们将采用 ETL 的术语来介绍将数据存入搜索引擎的过程。

因为我们知道是在和搜索引擎打交道，所以将更加侧重于搜索在 ETL 过程中的各个环节。如图 2.5 所示，这些环节包括提取（extraction）、充实（enrichment）、分析（analysis）和索引（indexing）。此处，提取是指从数据源获取文档的过程。充实

是可选的步骤，指的是将有助于相关性搜索的信息加入到文档中。分析，如我们在本章前面所看到的那样，是指将文档的文本或数据转化为可匹配的 token。最后，索引是指将数据存入上述数据结构的过程。

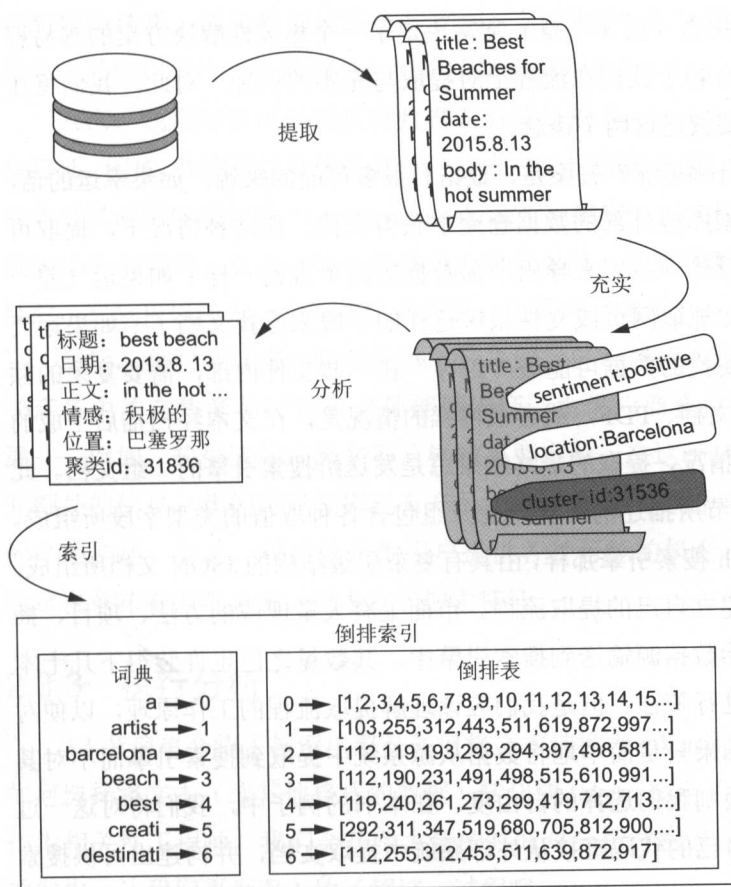


图 2.5 “搜索 ETL” 的完整管道 (pipeline)：提取、充实、分析和索引

我们将扼要介绍提取和充实。因为很多时候，这两个环节的具体细节完全依赖于源数据的存储方式。索引是与我们息息相关的，因为它在实现相关性搜索时与特征的启用或禁用息息相关。而分析，在相关性搜索中是最为重要的，此处我们将详加叙述，本书通篇对此也有多处讨论。回忆一下，分析过程将原生的文本和数据从文档转换成代表文档特征的 token。能够驾驭这些技术，以匹配来自用户查询的特征，直接关系到能否满足用户的信息需求。

### 2.3.1 将内容提取为文档

能否生成易于获取的文档可能与操纵搜索引擎的内部结构一样，在决定搜索的相关性方面都是非常重要的。尤其在本书的后续部分我们会看到，对内容的加工（第10章）和对字段的精心构造（第4~7章）常常决定了一个相关性解决方案的难易程度。这项工作的基础，有赖于我们在流程上对提取与充实的把控。对此，我们将在接下来的两个小节里简要叙述这两个概念。

我们搜索的文档来自哪里呢？答案是，数据有很多可能的来源。如果幸运的话，我们可以很容易地从数据库或外部的数据仓库中获得文档。在这种情况下，提取可能会比较简单，就如同设计一条用来导出所需数据的简单查询一样；如果运气差一些，也许就需要通过诸如抓取网页或文件系统这样的手段来寻找文档了；如果运气更差的话，也许会发现要找的数据可能被“封存”在一些文件内部，需要复杂的额外处理（例如 MS Word 文档、PDF，或者最糟糕的情况是，在文本经扫描后生成的图片中）。但是无论何种情况，提取的最终结果总是发送给搜索引擎的一组文档。此处的文档可能恰如 2.1.1 节所描述的那样，由一组包含各种取值的类型字段所组成。或者，就如同 Elasticsearch 搜索引擎那样，由具有复杂层级结构的 JSON 文档所组成。

我们的最终目标是建立自己的提取流程。市面上有大量现成的方法、项目、插件和产品能将数据从原始数据源输送到搜索引擎中。其数量之巨也许装得下几十本书的篇幅，此处我们不进行赘述。但是我们应该理解提取流程的工作原理，以便对文档的结构加以控制。如果只是简单地将数据从源系统中提取到搜索引擎而不对其结构进行处理，可能会限制我们选择的自由度。在本书的例子中，我们将对这一过程加以控制，利用我们自己的代码直接从外部系统中提取文档，并构建出可供搜索的文档来。

### 2.3.2 充实文档以清理、强化与合并数据

在充实阶段，来自提取环节的文档凭借额外的信息得到了强化。这对构建相关性搜索而言是一个重要的步骤，因为很多情况下提取得到的原生文档往往缺少足够丰富的特征来匹配用户的查询。文档充实包括三大部分：清理数据（cleaning data）、强化已有数据（augmenting existing data），以及合并外部数据（merging external data）。



首先，清理数据。如果我们想得到一流的搜索体验，那么花点时间去分析一下文档，从中找出像拼写错误和重复部分这样的低级错误，并纠正它们，这么做是非常值得的。否则，就有可能因为文档中包含搜索词的错误拼写，而导致用户搜索不到想要的结果。或者他们也许会找到同一篇文档的 20 个重复结果，这些重复的文档会将其他相关性文档挤到包含搜索结果的页面之外。

其次，我们时常可以对现有数据做后续处理（post-process）来强化已存在的那些特征。例如，我们可以利用机器学习技术对文档进行分类（classify）或聚类（cluster）。或者可以运用情感分析（sentiment analysis）来确定一篇文档中的文本在语气上是积极的还是消极的。总之，其可能性是无穷无尽的。当这种新的元数据被加入文档中以后，它就可以作为一个有价值的特征，供用户搜索了。

最后，我们可以将一些新的信息合并到来自外部数据源的文档中去。例如，电子商务中的在售商品时常来自外部供应商。由供应商提供的产品数据可能有所缺漏——例如，缺少诸如产品名称这样的重要字段。在这种情况下，就需要给文档补充额外的信息。我们可以利用已有产品的编号来查找产品名称，或者手工补充缺失的产品描述。总之，目的就是为用户提供各种可能的机会，找到他们需要的文档，而这也意味着需要更多、更丰富的搜索特征。

### 2.3.3 执行分析

2.1 节简单介绍了搜索引擎是如何将文本转换为 token 的。这一步是至关重要的。如何选择将文本（或其他格式的数据）转化成 token，决定了搜索引擎如何执行匹配。作为相关性工程师，我们会花费大量的时间对分析过程进行微调，以精确控制匹配的发生。让我们更为深入地了解这一过程吧。

当文档从其存储的地方被提取出来以后，经过可选的充实环节，最终被发送给了搜索引擎，在那里得到了分析处理。在分析过程中，搜索引擎对文档中的数据进行处理，将数据转换为 token，存储于搜索引擎的内部数据结构中。

如前所述，token 是代表文档中字段内容的一组符号（symbol）。在文本字段中，token 经常直接对应于单词。考虑这样一段文本“The Brown’s fiftieth wedding anniversary, at Café Olé”。根据配置，分析过程有可能会将这段文本划分成如清单 2.4 所示的 token。

### 清单 2.4 文本分词的例子

The, Brown's, fiftieth, wedding, anniversary, at, Café, Olé

不过通常, token 不再会是原来的单词, 而是经过标准化的、被过滤后的单词。上面的句子有可能会被切分成如清单 2.5 所示的样子。

### 清单 2.5 经标准化及过滤之后的文本分词的例子

brown, fiftieth, wedding, anniversary, cafe, ole

此处我们已经将单词转换成小写, 去掉了所有格符号和字母上的重音符, 并去掉了常用单词 (common words)。

不要以为 token 总得对应于单词; 任何数据类型都是可以被分词的。地理位置 (例如白宫的位置: 北纬 38.8977°, 西经 77.0366°) 也可以通过 geohash 进行分词。在本例中, 其合理的 token 结果有可能如清单 2.6 所示。

### 清单 2.6 利用 geohash 对地理位置进行分词

dqcjqcpee, dqcjqcpe, dqcjqcp, dqcjqc, dqcjq, dqcj, dqc, dq, d

这些 token 由对应地理位置的 geohash 字符串表示, 每一个 token 都是所在地理位置的精度由强到弱的一种表示 (第 4 章将讲述更多相关内容)。

搜索引擎的主要功能之一是 token 匹配。毕竟, 搜索引擎找到用户查询的文档, 所采用的方法就是 token 匹配。一方面, 来自文档的文本和其他数据经过分析——就是我们所说的分词——然后被存入倒排索引。另一方面, 查询条件也被分析并转化成 token。当文档中的 token 匹配查询中的 token 时, 文档就被认为是符合搜索的匹配项。

此处需要强调一下, 掌控分析的, 是作为相关性工程师的我们。改变将数据转换成 token 的方式会对搜索的相关性产生极大的影响。让我们以前面的例子作为示范。如果句子 “The Brown's fiftieth wedding anniversary, at Café Olé” 被切分成如清单 2.4 所示的样子, 并且我们向搜索引擎发起了一个查询, “fiftieth wedding anniversary”, 在这种情况下, 示例文档将被认为是匹配的。因为文档 token 和查询 token 是相同的。但是, 如果查询 “brown cafe ole”, 则会被认为是不匹配的, 因为清单 2.4 中的 token 包含了大写字母、所有格符号和重音符号, 而这些都不包含在查询 token 中。

**注意** 搜索引擎是很笨的：如果查询的 token 和文档的 token 不是每一个字节都完全一致，文档就不会被认为是匹配的！

这就是为什么我们说分析是非常重要的原因。为什么我们需要花费大量时间来标准化文本，因为这样一来，即使文档的原始文本与查询文本不完全相同，查询的 token 也能与文档相匹配。例如，来看一下改进后的分析，如清单 2.5 所示，当我们对例句进行标准化处理，去掉了大写、所有格符号和重音符之后，还是同样查询“brown cafe ole”，这次，与我们的示例文档完全匹配，三个词都匹配上了。

### 将 token 作为搜索特征

所有这些都与我们的搜索特征这一核心概念紧密相关。正如红色和圆形可以作为特征用来描述苹果一样，分析过程中得到的 token 可以作为特征用来描述文档。继续这一类比，如果我们要在一家杂货店中找苹果，就会寻找既是红色又是圆形的水果。搜索也是一样的：一位用户要寻找关于布朗结婚纪念的文档，他会将“Brown’s wedding anniversary”输入搜索引擎中，搜索引擎会依次进行分析查询，提取特征 (token)，然后尝试寻找拥有匹配特征的文档。这样大家就明白了，贯穿整个有关“搜索特征”的讨论，具体而言我们常常指的是“token”。我们使用“特征”一词是为了指明，token 起到了作为文档或查询的一种描述符的作用。

我们可以想象到，分析过程在将文本或其他值转变成 token 时给予了相关性技术工程师巨大的表现力。不过，常言道，能力越大，责任越大。良好的、描述性的特征在将查询与文档相匹配时是有帮助的，而无关的、甚至错误的特征则可能会导致文档无法被搜到！要确保我们的分析产生良好的特征，唯一的方法就是要对分析谙熟于胸。那么，让我们来看一下具体细节吧！

### 分析的组成

分析由三个步骤组成：字符过滤 (character filtering)、分词处理 (tokenization) 和 token 过滤 (token filtering)。让我们继续沿用之前的例子，逐一看一下每个步骤，并从头到尾演示一遍我们是如何对文本“The Brown’s fiftieth wedding anniversary, at Café Olé”进行分析的。在后面的章节中，我们会为大家示范如何对分析加以控制，不过在这里我们只介绍分析的流程和组成部分。

在第一步字符过滤中，有多种方法可以调整或过滤文本字段的字符。一个很好的例子就是 `HTMLStripCharFilter`，它以 HTML 作为输入，只返回包含在 HTML





图 2.8 分析——token 过滤

在我们讲索引之前，关于分析部分最后还有一点需要注意。在分析阶段，将一些额外的元数据信息与分析产生的 token 放在一起保存是一种很常见的做法。最为常用的元数据当属词位置和词偏移量了，它们在短语查询和高亮显示时都各有用处。我们也可以创建自定义的 token 过滤器，将各种元数据添加到 token 中被称为负载数据的位置。但值得注意的是：所有这些数据都可能会消耗掉大量的存储空间。如果我们对负载数据的使用还不是很熟悉，那么保守一点是比较明智的。关于这些内容的详细信息请参考表 2.1。（我们也会根据需要在本书的后续章节再次讲到这些内容。）

### 2.3.4 索引

当完成分析过程以后，我们将数据存入一种在 2.2 节中介绍过的称为倒排索引的数据结构中，这一过程被称为索引。尽管索引过程在技术实现上本身就是一个工程上的奇迹，但是在索引过程中我们的焦点往往集中在计算性能和资源管理方面，而非相关性。不过一些索引的策略有可能会对相关性产生影响——亦即，应该对哪些数据进行索引，以及应该采用哪些数据结构。

最重要的是，我们必须想好哪些字段需要索引或存储，而哪些字段则既需索引又需存储。一般意义上，索引是指将数据存入搜索引擎的过程。但是当我们讨论将字段数据存入核心数据结构中时，索引和存储都有各自更为具体的含义。这里的索引是指，利用提取到的 token 对倒排索引加以更新，以使相应字段可以被搜索到的过程。一个字段只有经过了索引才可以被搜索到。

存储指的是保留最初未经改动的文档内容，将其存入一种称为存储字段的数据结构中（见表 2.1），以便后面将其取出并放入搜索结果中呈现给用户。只有当数据



被存储了，搜索引擎才能将其返回并呈现给用户。作为一种优化的手段，一些工程师会存储尽可能少的信息，例如只存储像唯一标识这样最基本的信息。为了检索完整的文档，他们会再从外部存储设备获取字段的内容。而另一些人则为了方便起见，选择将数据存入搜索引擎，或者是减少对外部系统的依赖。存储数据也使搜索引擎得以对匹配结果进行高亮显示。正如我们前面提到的，这些高亮显示可以帮助我们解释为什么会有这样的匹配项，以及帮助用户更好地改进查询条件。

除了存储和索引数据，我们还可以选择使用或不使用表 2.1 中提到的许多其他数据结构。选用什么样的数据结构取决于搜索需求。我们将会在后续章节中讲到如何选择这些数据结构。

在索引阶段跟相关性有关的最后一点是，索引都是批量进行的。只有当分析完一定数量的文档或者等待了足够长的时间以后，经过分析的文档才会被提交给索引。文档只有被提交后才能被搜索到。正因如此，从发送文档给搜索引擎到通过搜索检索到文档，这之间会有一段时间延迟。我们在做相关性搜索时需要意识到这一点。所幸的是，在开发环境里，我们总是可以随时随地手工发起提交，因此不必担心延迟问题。而在生产环境里，Solr 和 Elasticsearch 都允许调整更新提交的设置，以满足我们的需求。这部分内容与相关性搜索并无直接关系，因此不在本书讨论范围之内。

## 2.4 文档的搜索和获取

当文档的 token、值以及原始内容被安全地存入搜索引擎的数据结构以后，我们终于可以开始搜索啦！在本节中，我们将先从文档匹配的机制开始，讲述有关搜索的基本内容。最后我们将以一段有关如何对匹配的文档进行评价、排序，并返回给用户的讨论结束本章。

### 2.4.1 布尔搜索：AND/OR/NOT

我们在 2.2.1 节中讲述了如何检索包含特定词的文档：首先从词典中查找词，找到对应的倒排表记录，然后就大功告成了。这些记录就是匹配的文档。

但是如果我们想匹配多个词该怎么办呢？布尔搜索（Boolean search）就是为此而设计的。布尔搜索可以合并多条查询的返回结果，从而更好地控制搜索结果。例如：（“shoe” AND “blue”）。让我们来看一看如何利用倒排索引这种数据结构中保存的



内容来实现典型的布尔操作（AND、OR 和 NOT）。

首先是 AND 运算符。还是以 2.2.1 节中使用的例子为例，考虑一下如何利用词典和倒排表找到既包含“shoe”又包含“blue”的文档。首先，我们需要为“shoe”和“blue”在倒排表中找到相应的记录，然后找到同时出现在这两个列表中的文档。因为倒排表的记录是一个有序的数字列表，因此求交集的算法非常简单，它的 Python 代码实现如清单 2.7 所示。

清单 2.7 布尔搜索 AND

```
def AND(term1postings, term2postings):
    term1doc = term1postings.next()
    term2doc = term2postings.next()

    matches = []
    while term1doc != None and term2doc != None :
        if term1doc == term2doc:
            matches.append(term1doc)
            term1doc = term1postings.next()
            term2doc = term2postings.next()
        elif term1doc < term2doc:
            term1doc = term1postings.next()
        else:
            term2doc = term2postings.next()

    return matches
```

利用两个词的倒排表记录集，准备好对每个记录集进行迭代

如果其中一个迭代结束，则循环结束

如果两个迭代指向同一篇文章，则此文档为匹配项

否则，其中一个向后递进

这里我们所要做的全部工作就是同时对两个倒排表记录集进行遍历，从两个列表的第一个文档开始，每次选择指向文档 ID 较小的那个列表，向后递进。任何时候，当所指的两个列表中的文档 ID 相等时，该文档就会被加入匹配项列表。若两个列表中的任何一个遍历结束，则算法终止并返回匹配结果。

从 AND 可以很容易地扩展到其他布尔运算。对于 OR 类型的搜索而言，我们返回的必须是两个列表中所有记录的合集，而非交集。对于 NOT 运算，我们根据匹配的一组文档，然后在所给的两个文档列表中，针对所有文档 ID 逐一判断每个文档 ID，去掉匹配文档（因为倒排表是经过排序的，所以这样的计算就变得相当简单）。

我们可以很容易地对布尔搜索进行扩展，执行复杂的、组合的布尔查询，以及多个字段的查询。要实现这一功能，我们不必做任何额外的工作。就组合查询而言，我们可以看到清单 2.7 中 AND 函数的输入参数和输出结果都是文档的 ID 列表，OR 和 NOT 运算符也同样。正因为如此，我们可以将布尔函数组合到一起，去执行任

意复杂的查询。扩展到多字段的查询也同样是很简单的，因为不论词位于哪个字段中，倒排表中的记录所指向的，都是使用相同 ID 的相同文档，因此来自任何字段和词的倒排表记录都可以用于前述算法。

## 2.4.2 基于 Lucene 搜索的布尔查询 (MUST/MUST\_NOT/SHOULD)

Lucene 有一种查询类型名叫 `BooleanQuery`，用于实现如前所述的行为。不过名字上有一点误导，行为上也不完全是我们所想象的那样。我们可能认为 `BooleanQuery` 是使用 AND、OR 和 NOT 运算符的——其实不是！它有三个子句，提供了相似的功能，但语义上稍有不同：SHOULD、MUST 和 MUST\_NOT。

- MUST 类型的子句要求文档中必须有匹配；否则，就视此文档为不匹配文档。
- SHOULD 类型的子句是指文档中可以存在匹配项，也可以不存在匹配项，但是符合 SHOULD 匹配条件的文档，其排名要比不符合该匹配条件的文档更靠前。
- 凡是符合 MUST\_NOT 匹配条件的文档一定不会被包含在搜索结果中，即使它满足 MUST 和 SHOULD 子句。

一个 `BooleanQuery` 可以包含任意数量的 SHOULD、MUST 和 MUST\_NOT 子句。但是如果一个查询不包含 MUST 子句，那么只有当文档满足一个或多个 SHOULD 子句时，它才会被认为是匹配文档。

在讲解例子之前，让我们来看一下 Lucene 的查询语法以及 `BooleanQuery` 的表现形式。Elasticsearch 和 Solr 用的就是这种语法来提供查询调试信息的，本书通篇用的都是这样的调试信息。在 Lucene 的查询语法中，MUST 和 MUST\_NOT 查询都是通过前缀来表示的。MUST 子句是 +，MUST\_NOT 子句是 -。SHOULD 子句不加前缀。来看一条简单的查询和一组文档：

查询：

```
black +cat -dog
```

文档：

- (a) my cat ran under the couch
- (b) black cats are mysterious
- (c) the dog scared the black cat

该查询是要找出任何必须 (MUST) 包含 cat, 应该 (SHOULD) 包含 black, 一定不能 (MUST\_NOT) 包含 dog 的文档。因此, 文档 (a) 和文档 (b) 被认为是匹配的, 因为它们包含了必要词汇 (required term) cat。在这两篇文档中, (b) 的排名会比 (a) 更高, 因为它包含了非必要词 (nonrequired term) black。文档 (c) 虽然同时包含了 black 和 cat, 但它不是匹配文档, 因为它包含了不被允许的词 (disallowed term) dog。

Lucene 的子句支持用圆括号进行分组。这里有一个简单的复合查询:

```
+(cat dog) black
```

该 MUST 子句是一个包含 cat 和 dog 的 SHOULD 子句的复合查询。文档要匹配, cat 或 dog 之一就必须包含在其中。由于 black 是 SHOULD 子句 (因为它没有前缀), 所以包含 black 的文档在搜索结果中的排名会更加靠前。

可能你还在好奇为什么 Lucene 要用这些奇怪的子句, 而非标准的布尔运算符 AND、OR 和 NOT。Lucene 的这些子句, 其更为“模糊化”的语义, 其实更利于搜索, 而标准的布尔运算符则是用来暗示集合在严格意义上的包含或非包含关系的。回想一下我们第一个查询的示例: black +cat -dog。如何用 AND、OR 和 NOT 来表现这个简单的查询呢? 答案是如下这样一个相当复杂的布尔查询:

```
(cat OR (black AND cat)) AND NOT dog
```

正如我们所见, Lucene 的语法更为简洁, 优势明显! 它往往能够帮助用户在表达上更加简练。在随后的几节中, 我们还将讨论 Lucene 是如何基于这些布尔查询为搜索结果进行排名的。

### 2.4.3 位置和短语匹配

单词的相对位置时常具有重要的语义价值。例如, “dress shoes” 的搜索结果应该返回黑色或棕色的男士皮鞋。搜索引擎如果不把词位置考虑在内, 仅仅搜索 “dress shoes”, 那么 we 也许就会发现搜索结果里充斥着女式服装 (women’s dresses) 和网球鞋 (tennis shoes) ——这当然不是用户期望的!

鉴于此, Lucene 提供了计入词位置的短语查询 (phrase query)。在 Lucene 的查询语法中, 短语查询是用引号引起来的, 例如, “dress shoes”。为了找到匹配的短语, 搜索引擎需要经过两个步骤:

- 找到匹配短语中每一个单词（dress 和 shoes）的所有文档。
- 去掉两个单词不相邻的那些文档（去掉 “this dress looks good with your shoes”，保留 “buy these handsome dress shoes”）。

短语查询返回的也是文档 ID 的列表；所以，短语查询兼容 Lucene 的 BooleanQuery。短语查询依赖于词位置（参见表 2.1）。而在 Elasticsearch 和 Solr 中，词位置默认情况下都是被包含于倒排索引中的。

有很多设置允许我们放宽短语查询对单词位置要求的约束，比如 phase slop。而且，值得注意的是，Lucene 有一个查询库名叫 span queries，它甚至允许我们更加精确地控制词的顺序和位置。本书将不会涉及这些高阶的位置查询（positional queries）。不过在这里提一下还是值得的，万一我们的搜索真的需要如此级别的复杂性呢。

#### 2.4.4 助力用户浏览：过滤、切面和聚合

人们在搜索大量文档时，对文档集合进行过滤，直到获得一个更为可控的集合，这样做往往是有好处的。例如要在亚马逊上购买一架尼康数码相机，我们无须关注电子产品以外的商品。而且，我们可能有自己的心理价位。如果你是业余摄影爱好者，也许就不会对 \$6,000 的尼康 D4S 感兴趣。基于前面提到的搜索引擎快速匹配文档的能力，以这样的方式对文档进行过滤是可行的。只是这里有一个重要的区别，因为亚马逊式的过滤方法通常是基于低基数（low-cardinality）字段<sup>1</sup>（诸如商品分类字段）或者是数值、日期型字段（例如，价格字段）的范围来完成的。

我们前面讨论过，切面（facets）给用户提供了一个搜索结果的自顶向下的视图。如图 2.9 所示，切面通常的展现形式是一组可供过滤的属性，加上每个属性搜索结果的数量统计。这样的用户界面可以帮助用户优化查询条件，快速理解搜索结果的集合，缩小相关性搜索的范围，以满足其需求。因此可以说，切面对用户而言是一种相关性的反馈。

---

<sup>1</sup> 指取值范围很小的字段。——译者注

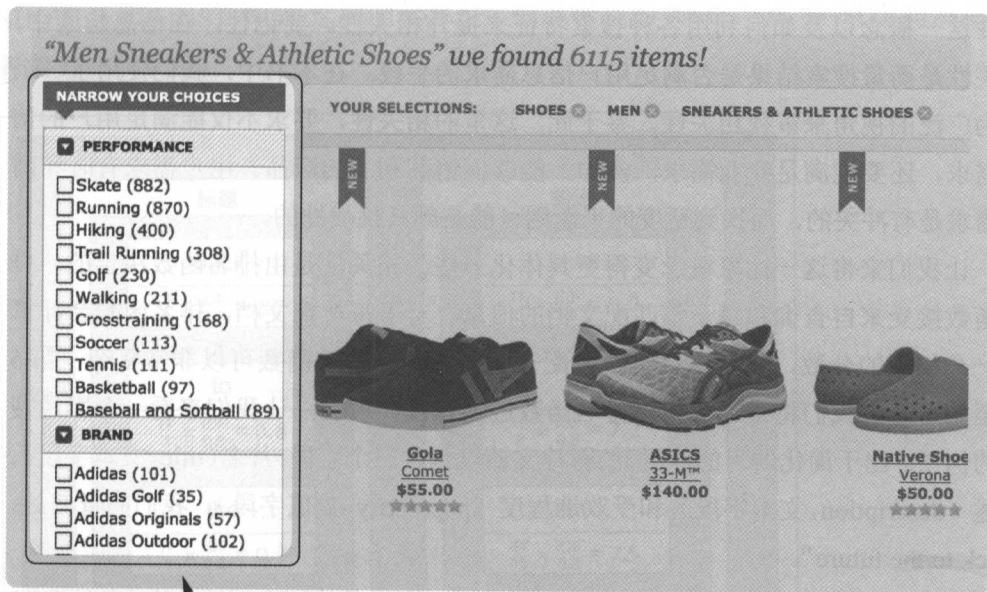


图 2.9 在 Zappos 的搜索结果页面上用切面进行搜索

Lucene 的数据结构支持复杂的、多层次的过滤、分组与聚合，而切面仅是这一强大威力的一小部分。幸运的是，Elasticsearch 有聚合功能（aggregations），允许用户基于某个字段的取值对数据进行过滤，对其他字段的取值进行分组，并最终对这些字段进行聚合处理（sum、mean、count、min、max 等），以此来实现高效的在线分析处理（online analytical processing，OLAP）。尽管聚合通常对使用者提出了较高的要求，但在搜索需要对文档进行切片（slicing）、取舍（dicing）和归总（summarizing）时，它所能提供的能力确实是绝无仅有的。

### 2.4.5 排序、结果排名，以及相关性

和许多其他的数据存储一样，搜索引擎支持按特定的顺序检索文档。其排序规则可基于浮点或整型字段的数值，或者是字符型字段的字母顺序。此外，排序规则还可以通过函数进行指定，在函数中我们可以利用一个或多个字段的数值来求得文档排序所依据的排序值。

但是在典型的搜索应用中，人们通常是不指定排列顺序的。相反，与搜索相匹配的文档是根据相关性由高到低的顺序返回的。在本章中，我们已经几次提到了相

关性这一概念以及如何利用各种搜索特征来提升相关性。要记住，在信息检索中，相关性是衡量搜索结果是否满足用户信息需求的手段。在本书中，我们使用了一个更为广泛的视角来审视相关性。鉴于此，这里的相关性，要求不仅能满足用户的信息需求，还要能满足商业需求，例如，能够促销高利润的商品。用户需求有时和商业需求是有冲突的，而找到适度的平衡则可能是颇具挑战性的。

让我们来将这一抽象概念变得更具体化一些。相关性是由排名函数决定的。排名函数接受来自查询和每一篇匹配文档的信息。对于每一篇文档，排名函数会计算出一个相应的分数，代表这篇文档匹配查询的程度。排名函数可以非常复杂；第5章至第7章，我们将讲述如何对排名函数进行修改。现在，让我们来看一个影片搜索的例子。出于简化的目的，我们的影片文档仅包含三个字段：片名(title, 文本字段)、描述(description, 文本字段)，和受欢迎程度(popularity, 数值字段)。我们的查询是：“back to the future”。

在对文档进行排名之前，我们首先必须通过对查询条件和文档的分析（参见2.3.3节）提取出 token，找到匹配的文档（参见2.4.1节）。可能还需要利用如前所述的用户自定义查询条件对其进行过滤。

有了匹配的文档，我们就可以用排名函数对其进行评价了。如图2.10所示，排名函数是具有层级结构的。

在图2.10的最里层，排名函数是基于查询词在某一字段中出现的频率（即在该字段中的词频）来计算评价值的。然后将结果乘以一个因子，该因子是基于词的常见程度（即该词在整个集合中的文档频率）以及字段所包含的词汇总数得到的。单词越少见、字段（包含的单词数）越小，则该因子就越大。对于我们的例句“back to the future”，任何在description字段中包含单词“future”的文档都会得到更高的评价，一方面凭借的是“future”出现的次数，另一方面则因为“future”在文档集中出现的频率相对较少。与之相反，包含单词“the”的文档会得到较低的评价，因为“the”是一个非常常见的单词。

再往上一层，同一字段中针对词的评价会被合并在一起。通常，这些词在Lucene的BooleanQuery中都属于SHOULD子句的一部分。并非所有的词都必须出现在要匹配的文档中，但是出现的词越多，文档的评价就越高。因此在我们的例子中，一篇文档同时包含单词“back”和“future”，其评价就要比只包含其中一个单词的文档高。SHOULD子句的这种直观的评价方式，是Lucene选择使用MUST、



SHOULD 和 MUST\_NOT 这样的运算符，而非标准的 AND、OR 和 NOT 布尔运算符的另一个原因。

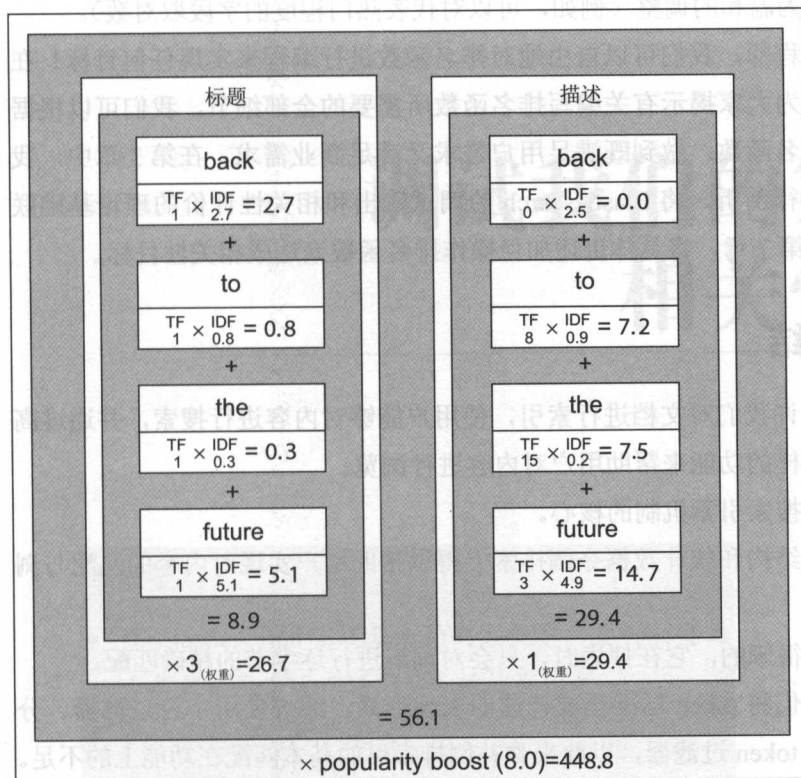


图 2.10 文档评价函数的例子（已简化）

通常，我们构造一条查询是想一次搜索多个字段。作为一名相关性工程师，我们可以通过放大字段的值（即增加字段的 boost 值）来表明它的重要性。例如，我们可能会对标题字段进行两次索引：一次是标准的分析，另一次则包含了常见的拼写错误。在本例中，当我们匹配拼写正确的字段时，其权重应该远远高于拼写有误的字段。所有字段的评价值随后会被合并在一起，相加求和，或者取最大值。

最后，相关性技术工程师还可以选择使用额外的乘法或加法形式的放大系数（boost 值），该放大系数可以是基于一个字段的数字值，也可以是基于函数的数字值，而该函数本身则依赖于一个或多个字段的取值。这就是搜索的“业务逻辑”时常发挥作用的地方。在我们的影片搜索的例子中，对近期热门影片进行放大处理也许是一种可取的做法，因为这些影片会比老电影卖得更好。因此，我们可以将最终的评

价值定义为所有字段分值的总和（见上一节）乘以代表热门程度的字段值。或者，如果这种放大处理的威力太过强大，我们也可以构造一个函数，根据热门的程度对最终的评价进行较为温和的调整（例如，可以对代表热门程度的字段取对数）。

作为相关性工程师，我们可以自由地对排名函数进行编程来实现任何目标！在本书中，我们将会为大家揭示有关编写排名函数所需要的全部细节，我们可以根据需要从容地调整排名函数，做到既满足用户需求又满足商业需求。在第3章中，我们会对评价技术进行剖析，将 Elasticsearch 的调试输出和相关性评价的理论基础联系起来。第5章至第7章，将具体讲述如何操作排名函数来达成相关性目标。

## 2.5 本章小结

- 搜索引擎允许我们对文档进行索引，使用户能够对内容进行搜索，并通过高亮或切面这样的功能来帮助用户对内容进行浏览。
- 倒排索引是搜索引擎机制的核心。
- 额外的数据结构和统计数据令倒排索引得以帮助用户实现对内容的匹配与浏览。
- 倒排索引是很笨的，它在搜索时，只会对词汇进行逐字节的精确匹配。
- 分析允许我们将 token 标准化成普通的表现形式，通过使用字符过滤器、分词器，以及 token 过滤器，以此来弥补倒排索引的基本匹配在功能上的不足。
- 分析使我们能够利用 token 来表达搜索中的重要特征。
- 布尔搜索是倒排索引这一数据结构最为基本的一种应用。
- 搜索的相关性结果是受排名函数控制的，同时也依赖于词频和文档频率等多种统计数据。
- 我们拥有绝对的控制力，可以通过分析来控制匹配（参见第4章），也可以控制排名函数（见第5章至第7章）！

# 调试我们的第一个 相关性问题的第3章

## 本章要点

- 用 Elasticsearch 实现基本的内容提取、索引和查询
- 分析解决没有返回预期结果的搜索问题
- 调试倒排索引的构建过程
- 分析解决相关性的 bug
- 解决我们的第一个相关性问题

第2章我们展示的是一个基于 Lucene 查询的有点理想化的蓝图。这一章，我们的搜索引擎出了点故障，大家会看到，我们是如何对一个真实的、运行中的搜索引擎进行调试的。什么工具可以用来查看搜索引擎的内部行为？为什么某些文档可以被查询出来，而其他更为相关的文档却不能？为什么那些看似无关的文档会排在相关文档的前面？

这一章为大家介绍的是初学者常常会遇到的问题。虽然解决方法一目了然，但为了解决这些问题，大家需要掌握调试相关性的方法。我们将使用这些技术去解决我们遇到的每一个相关性问题。就像在解决数学问题一样，将你所做的工作细节呈现出来也许是最为重要的一步。

大家将开始使用我们的搜索引擎，Elasticsearch，对一个真实的数据集进行搜索。当遇到这样的初学者常见的问题时，我们会把焦点放在调试相关性的两个主要的内部环节上：匹配和排名，这两点对相关性来说至关重要。当利用搜索引擎的调试能力对这些内容有了新的认识以后，我们就能根据那些最能描述内容的特征，开始使用搜索引擎来进行排名和匹配了。

通过本章的学习，我们将体验到相关性技术工程师“四处救火”的一天（如图3.1所示）。我们将分析并解决：为什么查询和排名不相匹配，为什么没有得到用户预期的结果，或者为什么一些奇怪的文档被认为比其他文档更具相关性。



图 3.1 作为一名相关性技术工程师，有几种工具可以帮助我们调试相关问题。

在开始体验之前，让我们先介绍几个主要的概念：搜索引擎、数据集，以及将要被用来操作相关性例子的开发环境。

### 3.1 Solr和Elasticsearch的应用： 基于Elasticsearch的例子

前一章介绍了基于 Lucene 的搜索引擎所包含的几个组成部分。我们应该使用哪个搜索引擎来举例呢？Solr？Elasticsearch？还是两者都用？

为了能够深入讲解，我们只选择了一种搜索引擎：Elasticsearch，来讨论我们的例子。以同样的深度同时涵盖两种搜索引擎，会令大家迷失其中，因为我们会无休止地去比较本质上相同但表面上却有所不同的那些配置细节。这些内容相当无趣（也可以说，与主题“毫不相干”），我们完全可以通过 Google 轻易地找到这些信息。

所幸的是，虽然表面上有所不同，但 Solr 和 Elasticsearch 在功能上是非常接近的。我们在本书中学到的内容对两种搜索引擎中的任何一种都是适用的。就像使用一本用 C 语言作为例子的算法书一样，其实我们可以用任何一种编程语言轻易地实现算法。这本书恰好用的是 Elasticsearch，但实际上我们可以用任何一种搜索引擎来实现本书中所讨论的那些相关性策略。

如果你是一名 Solr 开发人员，别担心：我们对 Elasticsearch API 的使用很基础，对你而言会感觉很熟悉。关于 Elasticsearch 中发生的细节，我们会给予恰如其分的解释，这样的话，即使你只对其中一种搜索引擎有所了解，也会帮助你轻松理解。此外，在本章中，为了替一些基本概念奠定基础，我们为 Solr 读者提供了一些建议。我们还提供了附录 B，帮助大家比较两种搜索引擎的功能。

特别强调一下，这不是一本介绍 Elasticsearch 的书籍。在使用搜索引擎的时候，我们所关心的是其与相关性有关的那些特性，而全然不会涉及其他特性或知识点，这些特性包括：内容分析、数据提取、特征缩放和性能表现。有很多不错的关于 Solr 和 Elasticsearch 的书籍或手册，如果你对这两个搜索引擎完全不了解，不妨在开始学习本书之前先去阅读一番。

## 3.2 最了不起的数据集：TMDB

在本书的大部分内容中，我们都将电影数据库（The Movie Database，TMDB）作为数据集。TMDB 是一个流行的在线电影和电视节目数据库。感谢 TMDB 允许我们使用它的数据集，我们也鼓励大家支持位于 <http://themoviedb.org> 的这一项目。我们很高兴地看到，TMDB 数据包含了许多搜索应用都必须支持的一些属性。当我们搜索影片时，这些属性包括：

- 长文本（如简介、摘要和用户评论等）
- 短文本（如导演、演员、片名等）
- 数值型属性（如用户评分、影片收入、获奖次数等）
- 影片发行时间和用于搜索的其他重要属性

在本书中，我们主要用到的是 TMDB 数据集的一个预先打包好的版本。位于本书在 GitHub 存储库中（<http://github.com/o19s/relevant-search-book>）的这个打包版本，是一个数据文件，它包含了本书写作期间所使用的一份 TMDB 影片数据的快照。该

文件，即 `tmdb.json`，是一个巨大的 JSON 字典。每一个条目都包含了一部影片的多个属性，比如片名和简介等。我们建议大家使用这份数据，因为这样得到的结果就会和本书的内容保持一致。当然你也可以直接使用 TMDB 的最新数据。在附录 A 中，我们介绍了如何对最新版本的 TMDB 数据建立索引。在该附录中，大家尤其会看到，我们是如何通过一个外部接口（API）把影片信息逐一提取出来，并加上演职人员信息的。

### 3.3 用 Python 语言编写的例子

当书中的例子需要少量编码时，我们会使用 Python，这是一种具有高可读性、命令式的语言，看起来很像伪代码。大家不必了解 Python 的语法（只要假装我们写的是伪代码就可以了）。我们不会使用任何语言相关的技巧，因此本书中的这些例子是很容易理解的。我们也尽量避免引入依赖（例如，甚至避免依赖一些非常优秀的 Elasticsearch 的客户端函数库）。相反，我们假定本书中的每一段 Python 代码，都只会用到如下所示的引用。此处我们引用的是 `requests`（一个 HTTP 的客户端函数库）和 Python 的 JSON 标准函数库。

```
import requests # requests HTTP library
import json # json parsing
```

请注意，基于代码可读性的考虑，我们所有访问 Elasticsearch 的例子中都使用了 `localhost` 和它的默认端口 9200。你可以在运行这些例子的时候根据自己的实际情况进行修改，指向你自己的 Elasticsearch 实例。

关于如何运行这些例子或访问 TMDB 数据的详细介绍，请参考本书在 GitHub 上的存储库（<http://github.com/o19s/relevant-search-book>）。该存储库包含本书中用到的所有例子和数据，在 README 文件中还有关于 Python、Elasticsearch 或所需函数库的详细安装说明。

### 3.4 第一个搜索应用

在开始之前，我们需要在 Elasticsearch 中为一些时下热门的影片所涉及的文本建立索引。在本章中，我们会非常详细地介绍我们要做的事情，并仔细地解释每一个步骤。为了避免在后续章节中出现过于冗长的介绍，我们将每一个模块都封装在



了一个 Python 的函数里。当完成对 TMDb 数据的索引，并发出了第一个查询指令之后，我们很快就会遇到相关性的问题，它将迫使我们不得不去调试这个看起来神秘莫测、行为古怪的搜索引擎。

要对影片信息进行索引，首先我们需要把它们都读进来！为了访问含有影片字典数据的 `tmdb.json` 文件，我们需要用到一个名为 `extract` 的函数。在清单 3.1 中，我们将通过对 JSON 文件的解析，得到每一部影片的信息，并把它们存入一个 Python 的字典里。

清单 3.1 从 `tmdb.json` 中提取影片信息

```
def extract():  
    f = open('tmdb.json')  
    if f:  
        return json.loads(f.read());
```

将JSON文件解析后  
放入Python字典；  
并返回该字典

那么返回的字典长什么样子呢？它是一个映射表，将 TMDb 中的影片 ID 映射到从 TMDb 中提取出来的影片信息。每一部影片都有许多记录其相关信息的字段。让我们来看一个例子。清单 3.2 所示的是记录在 Python 字典里的、关于影片 *Aquamarine*（中文名《美人鱼》）的一个片段。

清单 3.2 `tmdb.json` 中的 TMDb 影片范例

```
{  
    ...  
    "title": "Aquamarine",  
    "tagline": "A Fish-Out-Of-Water Comedy.",  
    "release_date": "2006-03-03",  
    "popularity": 0.340685029867431,  
    "original_title": "Aquamarine",  
    "budget": 12000000,  
    "cast": [  
        {  
            "name": "Emma Roberts",  
            "character": "Claire",  
            ...  
        },  
    ],  
    "vote_average": 5.6,  
    "runtime": 104  
}
```

影片名字

影片  
宣传词

影片演职  
人员列表

把一些有趣的数据加载进来以后，就可以利用 Elasticsearch 对这些文档进行索引了。Elasticsearch 有几种方式对文档进行索引。大多数情况下我们都会使用批量

索引 API (bulk index API)，因为它允许我们在一个 HTTP 请求中高效地对多个文档进行索引。不必过于担心批量索引 API 的输入与输出，因为本书的重点不是深入介绍这些索引 API。对于相关性搜索来说，至关重要的一点是能够按照分析和索引的新配置来重建索引。一旦索引被重建出来，我们就需要根据更新后的配置对文档进行再加工。

如前所述，我们需要建立一个 reindex 函数以便用于索引的重建。该 reindex 函数可以根据配置信息和 extract 函数返回的 movieDict 字典，重建 Elasticsearch 索引，并把对数据的索引放入 Elasticsearch，参见清单 3.3。

清单 3.3 利用 Elasticsearch 的批量索引 API 进行索引——reindex

```
def reindex(analysisSettings={}, mappingSettings={}, movieDict={}):
    settings = {
        "settings": {
            "number_of_shards": 1,
            "index": {
                "analysis": analysisSettings,
            }}
    }
    if mappingSettings:
        settings['mappings'] = mappingSettings

    resp = requests.delete("http://localhost:9200/tmdb")
    resp = requests.put("http://localhost:9200/tmdb",
                        data=json.dumps(settings))

    bulkMovies = ""
    for id, movie in movieDict.iteritems():
        addCmd = {"index": {"_index": "tmdb",
                           "_type": "movie",
                           "_id": movie["id"]}}
        bulkMovies += json.dumps(addCmd) + "\n" + json.dumps(movie) + "\n"
    resp = requests.post("http://localhost:9200/_bulk", data=bulkMovies)
```

禁用分片以消除对全局词统计的影响 ①

带有自定义分析和字段映射的索引设置

Elasticsearch 中有关字段分析方式的设置 (将在后续章节中介绍)

利用新的设置删除/重建 TMDB 索引 ②

③ 针对传入的 movieDict，对其中的每一部影片进行批量索引

在 reindex 函数中，我们首先与 Elasticsearch 进行交互，根据传入的配置信息为数据重建 tmdb 索引②。创建索引和在关系型数据库系统中创建一个数据库是类似的。索引中包含了文档信息，还有其他信息，如针对 tmdb 内容的搜索配置。当我们把 tmdb 索引作为一个整体进行操作时，就会用到 /tmdb 这一 Elasticsearch 的 HTTP 服务。

大家可能注意到①中传入的分片 (shards) 设置。在第 2 章中曾经说过，一个词的文档频率 (document frequency) 是排名结果的重要组成部分。它记录了一个词在整个索引中出现的次数。在分布式搜索引擎中，索引被物理地分成多个分片。文

档频率是按分片保存的。这样做，对于数据量较小的测试文档集而言，可能会导致结果排名出现问题。而对于较大的文档集，分片的影响通常是较为均衡的。为了测试的可重复性，此处我们将禁用分片。

从❸开始，我们利用批量索引 API，开始构造一个批量索引命令的字符串，并将其传给 Elasticsearch。这里的 `addCmd` 告诉 Elasticsearch，我们要对文档进行索引。我们需要告诉 Elasticsearch 关于每篇文档的一些元数据，包括其所保存的位置（`_index: tmdb`）、类型（`_type: movie`）以及唯一标识 ID（取自 TMDB 的 ID）。随后的一行，我们将该文档追加到后面用于索引。在该行中，我们把命令和文档都追加到了 `bulkMovies` 字符串的末尾，以便能执行索引操作。对 `movieDict` 中的每一部影片都重复这个过程。当一个完整的批量索引命令构造完毕之后，我们发起一个 POST 请求，将这个巨大的 `bulkMovies` 字符串发往 Elasticsearch 的 `/_bulk` 服务。

综上所述，我们终于可以对影片信息进行索引了。结合使用 `extract` 和 `reindex`，可以利用清单 3.4 中的代码，将数据从 TMDB 添加到 Elasticsearch 中。

#### 清单 3.4 把数据从 TMDB 添加到 Elasticsearch 中

```
movieDict = extract()
reindex(movieDict=movieDict)
```

可喜可贺！我们已经构建出自己的第一个 ETL（提取、转换、加载）管道了。到目前为止，我们已经完成了如下工作：

- 从外部系统中提取信息。
- 把数据转换成满足搜索引擎要求的格式。
- 对数据进行索引并存入 Elasticsearch。

此外，通过在 `reindex` 函数中利用命令告知 Elasticsearch 新的索引（`_index: tmdb`）和新的类型（`_type: movie`），我们新建了一个索引（非 SQL 数据库）和一个文档的类型（非 SQL 表）。接下来，当我们想进行搜索或与 `tmdb` 的索引进行交互时，就可以使用 `tmdb/movie/` 或 `tmdb/` 这样的 Elasticsearch URL 路径。

### 3.4.1 针对 TMDB Elasticsearch 索引的第一次搜索

现在我们可以开始搜索了！对于这个电影应用而言，当用户在搜索栏中输入搜

索内容后，我们需要解决系统如何响应的问题。要实现它，需要使用 Elasticsearch 的领域特定查询语言（Query domain-specific language, DSL）或者 Query DSL。

Query DSL 告诉 Elasticsearch 如何以 JSON 格式来进行搜索。此处，我们需要指定一系列因子，比如“必须包含某文本”（required）的子句、“不应包含某文本”的子句、放大系数（boost 值）、字段权重、评价函数，以及其他一些可以控制匹配和排名的因子。我们可以将 Query DSL 看作搜索引擎的 SQL，这种查询语言是专门针对扁平结构、非标准化文档的排名检索的。

作为一名相关性技术的新手工程师，我们将从 Elasticsearch 的 multi\_match 查询的一个简单应用入手。multi\_match 就像是 Elasticsearch 的一把瑞士军刀，可以构建针对多个字段的查询。因为大部分搜索问题都涉及多字段搜索，因此很多人都是从这里入手来寻求相关性搜索的解决方案的。通常，相关性搜索的解决方案都是先试图构建一个 multi\_match 的查询，列出带有放大系数（以符号 ^ 指定）的待搜索字段。所谓放大（boost）是指这样一种行为，它可以为相关性的评价价值加上或乘以一个常量因子、查询，或者函数。在本例中，放大系数的使用很简单；我们只是将片名（title）的评价价值放大了 10 倍，从而让搜索引擎了解到该字段的相对重要性。

我们来实现一个 search 函数，它可以让我们利用传入的 Query DSL 进行搜索。函数 search 的功能一目了然，我们传入一个 query 对象，然后打印出按相关性排序的搜索结果，代码如清单 3.5 所示。

清单 3.5 search 函数

```
def search(query):  
    url = 'http://localhost:9200/tmdb/movie/_search'  
    httpResp = requests.get(url, data=json.dumps(query))  
    searchHits = json.loads(httpResp.text)['hits']  
    print "Num\tRelevance Score\t\tMovie Title"  
    for idx, hit in enumerate(searchHits['hits']):  
        print "%s\t%s\t\t%s" %  
            (idx + 1, hit['_score'], hit['_source']['title'])
```

利用传入 query 的 Query DSL 来执行既定的搜索

打印输出搜索结果

我们传入的 Query DSL 查询又是长什么样子的呢？在清单 3.6 中，我们利用 multi\_match 构造了一条 Query DSL 的搜索语句。试图告诉 Elasticsearch，片名字段比简介字段在排名时重要 10 倍<sup>①</sup>。在本章中，我们会对这样的尝试是否有效做出评估。

**对 Solr 读者的建议** Solr 用的不是 multi\_match，它推荐大家从

“dismax”系列的查询解析器入手。对 Solr 用户而言，其查询也许是这样开始的：`http://localhost:8983/solr/tmdb/select?q=basketball with cartoon aliens&defType=edismax&qf=title^10`。请注意，虽然出发点相同，但该查询和 Elasticsearch 的 `multi_match` 查询解析器相比，在表现上是有所不同的。可以查看第 6 章及附录 B 来了解更多详细信息。

清单 3.6 所示的是我们利用 Query DSL 的第一次“hello world”搜索。

### 清单 3.6 我们的第一次搜索

```
usersSearch = 'basketball with cartoon aliens'
query = {
  "query": {
    "multi_match": {
      "query": usersSearch,
      "fields": ["title^10", "overview"],
    }
  }
}
search(query)
```

用户的搜索  
关键词

把“title”的重要性  
相比于“overview”  
① 放大10倍

输出：

Num	Relevance Score	Movie Title
1	0.8424165	Aliens
2	0.5603433	The Basketball Diaries
3	0.52651036	Cowboys & Aliens
4	0.42120826	Aliens vs Predator: Requiem
5	0.42120826	Aliens in the Attic
6	0.42120826	Monsters vs Aliens
7	0.262869	Dances with Wolves
8	0.262869	Interview with the Vampire
9	0.262869	From Russia with Love
10	0.262869	Gone with the Wind
11	0.262869	Fire with Fire

哦，不——这样的搜索结果是不行的！从查询关键词“basketball with cartoon aliens”我们可以推断出，用户很可能想找的是 *Space Jam*（中文名《空中大灌篮》）——一部关于 Looney Tunes（卡通人物）得到 Michael Jordan（迈克尔·乔丹）的帮助，在一场篮球比赛中反击外星人的电影。看起来用户似乎并不知道影片的名字，他在尝试通过一个描述性的查询来进行搜索——这是一个常见的用户使用场景。遗憾的是，大部分排名靠前的影片看起来和篮球或外星人有关，但并不是和两者都相关。而其他影片则看起来和篮球或外星人毫不相干，我们完全被这样的结果搞晕了，

*Space Jam* 在哪里呢？如果请求从 Elasticsearch 返回更多的内容，最终我们会看到想要的结果：

43 0.016977157

Space Jam

为什么看似无关的影片被搜索引擎认为更有价值呢？我们如何诊断问题并找到解决方案呢？作为相关性技术工程师，搞明白为什么搜索引擎返回如此奇怪的结果是我们的家常便饭。我们需要回答两个主要问题：

- 为什么某些文档能够匹配查询条件？为什么像 *Fire with Fire*（中文名《火线反击》）这样的影片都能匹配我们的查询呢？
- 为什么相关性较低的文档的排名这么高？为什么 *The Basketball Diaries*（中文名《边缘日记》）的排名会比 *Space Jam* 还高呢？

我们想要很快找到问题的答案。时间在一分一秒地过去，用户的搜索体验依然糟糕。

## 3.5 调试查询匹配

对于“basketball with cartoon aliens”这条失败的搜索，问题可能出在哪儿呢？要寻找问题的答案，首要的也是最基本的方法，就是对查询的词匹配（term matching）行为进行调试。在我们的工作中，经常会遇到一篇相关的文档本该匹配却没有匹配的情况。相反，我们可能会感到惊讶，当低价值的或错误的词匹配发生时，无关的文档会出现在查询结果中。即使在检索到的这些文档中，匹配或不匹配某个词也可能会对相关性排名产生影响——要不就是因为不合理的匹配导致排名偏高，要不就是因为意料之外的错误导致排名偏低。我们需要借助 Elasticsearch 针对分析和查询的验证调试工具来介入这一过程。

首先，要提醒大家我们所谓的匹配是什么含义。回忆一下第2章中所给的定义：在倒排索引中，匹配指的是一种严格而精确的二进制意义上的相等。搜索引擎不会聪明到能够理解“Aliens”和“alien”表示的是一个意思，或者知道“extraterrestrial”也指的几乎是同一个意思。说英语的人明白，这些说法都是外星人的意思；或者，正如我们之前所讨论的那样，这两个词指向的特征就是“alien-ness”。但是对于愚钝的搜索引擎而言，这两个 token 是以不同的 UTF-8 二进制字符串的形式存在的。两个字符串 0x41,0x6c,0x69,0x65,0x6e,0x73 (Aliens) 和 0x61,0x6c,



0x69,0x65,0x6e (alien), 完全不同, 也不相匹配。

这种严格的匹配行为涉及两个不同的方面：

- 查询解析 (query parsing) —— 我们的 Query DSL 查询是如何被搜索引擎翻译成针对字段的、特定词语的匹配策略的。
- 分析处理 (analysis) —— 根据查询和文档的文本来生成 token 的过程。

理解了查询解析, 我们就能够准确理解自己的 Query DSL 查询是如何利用 Lucene 的数据结构来满足针对不同字段的搜索行为的。通过分析, 我们就有希望能对文本加以揣摩、询问、窥探、侦测, 从而把隐藏在文本中真正表达出特征 “alien-ness” 的词提取出来。我们还可以进一步识别出那些无意义的词, 比如有些词, 虽然可以匹配上, 但其所表达的特征却无足轻重, 这样的低价值词汇, 会产生不合理的匹配。

只有理解了底层数据结构是如何被创建和访问的, 我们才有希望掌控整个过程。让我们一起来分析一下前面的搜索, 看一看匹配所出现的问题, 是否是由于不经意间包含了如 *Fire with Fire* 这样不合理的匹配影片而导致的。

### 3.5.1 检查底层查询策略

为了检查 Elasticsearch 是如何进行匹配的, 我们首先要做的事情, 是弄清楚查询是如何被解析的。这一过程会将我们的搜索查询拆解成另一种描述形式, 这样可以更好地实现与 Lucene 数据结构的底层交互。要做到这一点, 我们会用到 Elasticsearch 的查询验证服务。从清单 3.7 所示的代码示例中可以看出, 该服务以一条 Query DSL 查询作为输入参数, 返回的是一种低层次的解释信息 (low-level explanation), 用以揭示搜索引擎为满足查询所采取的策略。

**对 Solr 读者的建议** 为了得到同样的查询解析调试信息, 需要在 Solr 的查询中加上参数 debugQuery=true。这样我们就能在 Solr 响应结果中的 parsedquery 部分看到和 Elasticsearch 查询验证服务类似的输出了。

#### 清单 3.7 对查询行为进行解释

```
query = {
  "query": {
    "multi_match": {
      "query": usersSearch,
      "fields": ["title^10", "overview"]
    }
  }
}
```

```

    }
}
httpResp = requests.get('http://localhost:9200' +
                        '/_validate/query?explain',
                        data=json.dumps(query))
print json.loads(httpResp.text)

```

← 针对验证API  
发起验证请求

响应结果：

```

{u'_shards': {u'failed': 0, u'successful': 1, u'total': 1},
 u'explanations': [
 {u'explanation':
  u'filtered((((title:basketball title:with
  title:cartoon title:aliens)^10.0) |
  (overview:basketball overview:with
  overview:cartoon overview:aliens)) -> cache(_type:movie)',
  u'index': u'tmdb',
  u'valid': True}],
 u'valid': True}

```

← 系统是如何按照Lucene的查询语法执行查询的

此处返回的解释字段（见粗体部分）列出了我们感兴趣的内容。我们的查询条件被翻译成了一种更为精确的语法，通过它，我们可以获得更多深入的信息，理解Lucene是如何处理我们的Elasticsearch查询的。

```

((title:basketball title:with title:cartoon title:aliens)^10.0) |
(overview:basketball overview:with overview:cartoon overview:aliens)

```

### 3.5.2 剖析查询解析

查询验证服务所返回的，是我们的Query DSL查询的另一种表现形式，它可以帮助我们进行匹配问题的调试。让我们来看一下它的语法；其实在第2章中已经介绍过这种语法的基本知识。查询验证的输出内容让人立刻就会联想到Lucene的查询语法<sup>1</sup>——一种底层的、精准的用来定义搜索的方法。由于它格外精准，因此用Lucene查询语法来描述一个相关文档的需求，能够更好地反映出Lucene自身是如何利用倒排索引来执行搜索的。

正如我们在第2章中所讨论的，Lucene的查询语句是由一系列布尔子句组成的，如MUST(+)、SHOULD和MUST\_NOT(-)。每一个子句都需要指定一个在底层

<sup>1</sup> 实际上，其表现形式在Java中依赖于每个Lucene查询的toString方法，它会尝试使用（但不一定总会准确体现）严格的Lucene查询语法。

文档中进行搜索的字段，且要满足 `[+/-]<fieldName>:<query>` 这样的格式。对于匹配过程的调试来说，这一子句中最为重要的部分就是匹配本身，即 `<fieldName>:<query>`。如果看一下前面例子中出现的子句，比如 `title:basketball`，我们会发现，这是要在 `title` 字段中查找单词 `basketball`。每一个子句都是一个简单的词查询，也就是一个在倒排索引中的词搜索。除了词查询，我们能遇到的最多的就是短语查询。这在第 2 章中也讨论过。在 Lucene 的查询语法中，短语是要用引号括起来的，如 `title:"space jam"`，表示引号中的两个词必须是连在一起的。

在我们的例子中，当我们将目光移到匹配表达式的上一层，就能看出 Lucene 的查询策略了。虽然我们现在关注的是匹配问题，但它包含的信息并不只是这些。在最里层的匹配表达式之上，可以看到 4 个 `SHOULD` 子句被放在了一起（用圆括号括住）进行评价：

```
(title:basketball title:with title:cartoon title:aliens)
```

用因子 10 加以放大（这是我们查询时要求的）后，我们就得到了下面的表达式：

```
(title:basketball title:with title:cartoon title:aliens)^10
```

和另一条查询相比，取两者之间评价价值最大者（`|` 符号），就得到了下面的表达式：

```
((title:basketball title:with title:cartoon title:aliens)^10.0) |  
(overview:basketball overview:with overview:cartoon overview:aliens)
```

继续阅读本书，我们还会介绍这一伪 Lucene 查询语法的其他部分。

奇怪，有很多杂七杂八的评价都冒出来了，让人觉得十分意外。在本章后面部分，我们将对评价过程进行更为深入的调试；不过目前最要紧的是，利用词查询所给的信息，弄明白为什么甚至像 *Dances with Wolves*（中文名《与狼共舞》）或 *Fire with Fire* 这样不合理的匹配也会被认为是满足匹配条件的。

### 3.5.3 调试分析，解决匹配问题

现在我们已经知道搜索引擎是用哪些词来进行搜索的了，接下来需要对匹配部分进行调试，看看文档是如何被分解成一个个的词语，然后放到索引中的。毕竟，如果搜索的关键词在索引中不存在，那么我们的搜索就会失败。前面我们已经举过这样一个例子了。不管我们的直觉是什么，对单词 `Aliens` 的搜索是不会与 `alien`

相匹配的。而且，对关键词的搜索有可能会得到不合理的匹配结果，这些匹配没有提供任何有价值的信息。例如，在英语中，单独匹配 the 是没有意义的。对于我们那些接受过英语训练的用户而言，它并没有指明隐藏在文本中的重要特征。

尽管我们对文档应该如何拆分词语有自己的直觉，但 Elasticsearch 的分析机制经常会让我们感到惊讶。搜索引擎的分析过程是一个我们经常需要去调试的环节。我们已经知道了这些关键词是如何提取的：通过索引阶段的分析过程来完成。分析器（analyzer）是一种定义分析流程的实体。它包含了我们在第2章中已经讨论过的一些组件，如：字符过滤器（character filters）、分词器（tokenizer）和 token 过滤器（token filters）。在 Elasticsearch 中，我们可以在许多不同的层面指定对分析器的使用，包括：索引（针对整个 TMDb 的）、节点（针对一个 Elasticsearch 的运行实例）、类别（针对所有影片）、字段，甚至在查询阶段的某一条特定的查询。如果我们还没有指定分析器，则会使用默认的标准分析器。我们可以利用所掌握的这一知识，以及 Elasticsearch 的分析服务，来查看文档中的文本是如何被转成 token，并最终形成倒排索引的。

或许，如果我们看了对片名 *Fire with Fire* 的分析结果被加入倒排索引的过程，就可能知道与查询相匹配的词应该是什么了，参见清单 3.8。那么，我们也许就会明白为什么看似随机、无关的影片会出现在搜索结果中了。

**对 Solr 读者的建议** 虽然 Solr 有一个类似的 API，但是它还提供了一个功能丰富的管理界面，里面包含了一个分析调试工具。在 Solr 的管理界面中选择我们所使用的“core”，然后选择“analyzer”可执行类似的调试。

### 清单 3.8 调试分析

```
resp = requests.get('http://localhost:9200/tmdb/_analyze' \
                    '?analyzer=standard&format=yaml',
                    data="Fire with Fire")
print resp.text
```

请求利用标准分析器对字符串“Fire with Fire”进行分析

结果如下所示（采用 YAML 格式做了美化处理）：

```
tokens:
- token: "fire"
  start_offset: 0
  end_offset: 4
  type: "<ALPHANUM>"
```

位于 token 流中的一项，给出了从一个 token 中取得到的属性

起止偏移量表示 token 在原文中所处的位置

```

position: 1
- token: "with"
  start_offset: 5
  end_offset: 9
  type: "<ALPHANUM>"
position: 2
- token: "fire"
  start_offset: 10
  end_offset: 14
  type: "<ALPHANUM>"
position: 3

```

position表示词的顺序、距离，以及邻接情况

从输出中我们可以看到经标准分析器从“Fire with Fire”中提取到的每个 token 所包含的重要属性。通过分析得到的这个 token 列表也被称作 token 流。在该 token 流中，我们提取出三个 token：fire、with 和 fire。请注意，文本是用空格进行分隔并被转成了小写的。此外，除了 token 文本，还有很多其他属性被加了进来。请注意 offset 的值，它表示每一个 token 在原文中确切的字符位置；还有 position 的值，它表示 token 在 token 流中的位置。

进行分析之后，我们对 token 流进行了索引并将其放入倒排索引中。为了调试和讲解，我们可以使用一种被称作“SimpleText”<sup>1</sup>的数据结构来展现倒排索引，它是由 Mike McCandless 创造的一种纯用作教学目的的索引存储格式。我们可以采用这一格式来理解和分析倒排索引的结构。

仅以单词 fire 为例，让我们花一点时间来看一下，前述的 token 流是如何被翻译成以 SimpleText 形式表达的索引的，如清单 3.9 所示。

### 清单 3.9 单词 fire 以 SimpleText 形式表达的索引

```

field title
term fire
  doc 0
    freq 2
    position 1
    position 3
  doc 2
  ...

```

表示 doc 0 包含单词“fire”

搜索引擎在索引期间的目标就是要把 token 流中的信息放入倒排索引，并将文档放入其所适合的词下面。在对某个 token 出现的次数进行统计之后（本例中 fire

<sup>1</sup> 可以阅读 Mike McCandless 有关 SimpleText 的博客，以获取更多的信息：<http://blog.mikemccandless.com/2010/10/lucenes-simpletext-codec.html>。

出现了两次), 索引会把这些信息都添加到单词 `fire` 的倒排表中。在 `fire` 下面, 添加文档 `doc 0` ❶。然后再把 `fire` 在 `doc 0` 中出现的次数存成 `freq`, 并通过多个 `position` 条目记录下每一次该词出现的位置。待所有 `token` 都得到处理后, 该文档就会被添加到 `fire` 和 `with` 这两个词的倒排表记录中, 如清单 3.10 所示。

清单 3.10 title 字段的索引情况, `Fire with Fire` 被高亮显示

```
field title
term fire
  doc 0
    freq 2
    position 1
    position 3
  doc 2
  ...
term with
  doc 0
    freq 1
    position 2
  ...
```

← 位于“title:fire”倒排表记录下的“Fire with Fire”

← 位于“title:with”倒排表记录下的“Fire with Fire”

正如我们前面所讨论的, 消费 `token` 流的是那些数据结构, 而非倒排索引。在 `Lucene` 中有许多数据结构可供选择。为了实现我们的目标, 我们应该考虑那种能够解析 `token` 流、提供有关词汇的全局统计信息 (`global-term statistics`) 的数据结构, 如文档频率。在本例中, `fire` 的文档频率会在原有基础上加 1, 表示一篇新文档的加入。

请注意, 我们是可以深入控制这一过程的, 这一点很重要。对分析过程的控制通常是逐字段进行的。我们可以利用第 2 章中讨论过的组件, 如: 字符过滤器、分词器和 `token` 过滤器, 来为我们的字段定义自己的分析器。但首先, 在了解了查询以及索引中的词以后, 我们得看一下, 为什么像 `Fire with Fire` 这样不合理的查询结果竟然会排在第一位。

### 3.5.4 比较查询条件和倒排索引

现在, 我们已经准备好了, 将经过解析的查询条件与倒排索引的上下文进行对比。如果我们把经过解析的查询条件

```
((title:basketball title:with title:cartoon title:aliens)^10.0) |
(overview:basketball overview:with overview:cartoon overview:aliens)
```

和根据“`Fire with Fire`”的 `token` 流获得的倒排索引片段进行比较, 就能清楚地看到



匹配发生的地方：

```
field title
  term fire
    doc 0
      freq 2
      position 1
      position 3
    doc 2
    ... (more docs)
  term with
    doc 0.
      freq 1
      position 2
    ...
```

在倒排表中，“with”和“fire”处于同等重要的位置

子句 `title:with` 找到了 `doc 0`，也就是倒排索引中的“Fire with Fire”。回想一下单词匹配是如何工作的，我们就能理解此处的机制了。我们的文档位于索引中的 `with` 下面。于是，它与其他同样匹配 `with` 的文档一起出现在了搜索结果中。正如我们在前面一章所讨论的，这纯粹是一个机械化的过程。当然，对于讲英语的人来说，按照 `with` 进行匹配是没有什么帮助的，这会让他们感到困惑，为什么像这样没有实际意义的单词（noisy word）在匹配时会被认为是很重要的呢。

其他不合理的匹配影片看起来也都属于这一类。比如 *Dances with Wolves* 或 *From Russia with Love*（中文名《俄罗斯之恋》）这样的电影，它们被轻松地排在了搜索结果中靠前的位置，就如同那些与 `basketball` 或 `aliens` 这样的重要单词相匹配的影片一样。在没有帮助的情况下，搜索引擎无法区分哪些是有意义的、有效的、重要的英文单词，而哪些是没有实际意义的、价值不高的单词。

### 3.5.5 通过修改分析器来修正我们的匹配

幸运的是，这个匹配问题有一个很简单的解决办法。我们曾嘲笑 Elasticsearch 不懂英语。但实际上，Elasticsearch 有一个能够很好地处理英文文本的分析器。它把字符过滤器、分词器和 `token` 过滤器串在一起，将英文转化成单词的标准形式。它能找到英文单词的词根（`running` → `run`），并去掉那些无意义的单词，比如 `the`，也被称之为禁用词（stop words）。幸运的是，`with` 就是这样的一个禁用词。把它从索引中删除就能够解决我们的问题。

该如何使用这个分析器呢？很简单：我们需要为字段指定一个不同的分析器。因为改变索引阶段的分析方法会对倒排索引的结构产生影响，所以我们需要重新对

文档进行索引。为了定制分析方法，我们需要重建索引，并重新运行之前构建索引的代码。主要的不同点在❶处，如清单 3.11 所示；我们需要在创建索引之前，明确为字段配置英语分析器。

**对 Solr 读者的建议** Solr 的 schema.xml 文件可以用来对 Solr 字段进行配置。与字段的 fieldType 相关联的 analyzer 标签，决定了该字段采用何种分析器。默认情况下，schema.xml 定义了一系列字段类型，包括 text\_en，它很适合处理英文文本。改变分析器以及字段的设置需要像 Elasticsearch 那样对 Solr 重新进行索引。

清单 3.11 用英语分析器重新进行索引

```
mappingSettings = {
  "movie": {
    "properties": {
      "title": {
        "type": "string",
        "analyzer": "english"
      },
      "overview": {
        "type": "string",
        "analyzer": "english"
      }
    }
  }
}
movieDict = extract()
reindex(mappingSettings=mappingSettings, movieDict=movieDict)
```

❶ 修改 “title” 和 “overview” 字段来使用英语分析器

根据新的字段映射来重建索引

太棒了！这样做有用吗？让我们重新对 “Fire with Fire” 进行分析，看一看结果：

```
resp = requests.get('http://localhost:9200/tmdb/_analyze?field=title&format=yaml',
                    data="Fire with Fire")
```

响应结果：

```
tokens:
- token: "fire"
  start_offset: 0
  end_offset: 4
  type: "<ALPHANUM>"
  position: 1
- token: "fire"
  start_offset: 10
```

“with” 不再出现于 token 流中

```
end_offset: 14
type: "<ALPHANUM>"
position: 3
```

第二个“fire”  
的position值没  
有变化

请注意，在 token 流中已经没有 with 了。尤其我们注意到，position 1 和 3 之间的间隙。Elasticsearch 正是通过这种位置上的间隙来反映 token 的删除的，从而避免了不合理的短语被匹配上。重新执行查询验证我们发现，with 也同样从查询中被删除了。

```
{u'_shards': {u'failed': 0, u'successful': 1, u'total': 1},
 u'explanations':
  [{u'explanation':
    u'filtered((((title:basketbal title:cartoon title:alien)^10.0) |
      (overview:basketbal overview:cartoon overview:alien)))
    ->cache(_type:movie)',
   u'index': u'tmdb',
   u'valid': True}],
 u'valid': True}
```

新的查询策略。注意，按照英文词根处理的规则，“basketball”现在被转化成了词根“basketbal”

的确，匹配的结果已经非常接近我们的要求了。至少已经都与 aliens 有关了。而且，由于经过了精细的分析、词根的提取、token 的标准化等处理，那些之前曾经遗漏了的，涉及 alien 的其他匹配，也被纳入进来。

Num	Relevance Score	Movie Title
1	1.0643067	Alien
2	1.0643067	Aliens
3	1.0643067	Alien3
4	1.0254613	The Basketball Diaries
5	0.66519165	Cowboys & Aliens
6	0.66519165	Aliens in the Attic
7	0.66519165	Alien: Resurrection
8	0.53215337	Aliens vs Predator: Requiem
9	0.53215337	AVP: Alien vs. Predator
10	0.53215337	Monsters vs Aliens
11	0.08334568	Space Jam

恭喜你！通过使用英语分析器，我们已经往前跨越了一大步。现在我们将目标放在第 11 条记录上面。通过基于英文的简单分析，我们已经得到了一系列合乎情理的文本映射，对应于文本的“alien-ness”特征。我们还去除了一些理应与任何文本特征都不相关的单词，即：禁用词。

在后面的章节中，我们会探讨更多的用户案例，需要对 token 的表现形式进行调整，比我们此处所做的还要更加深入。因为词（term）和文本特征相类似，所以将文本翻译成 token 经常需要根据领域的不同而进行深入定制。现在，我们准备开

始调试“搜索反应式 (search equation)”的下一个阶段：相关性排名。

### 3.6 调试排名

在解决了匹配问题之后，我们仍然心存疑问，为什么像 *Alien*（中文名《异形》）、*Aliens*（中文名《异形2》）和 *Basketball Diaries* 这样的电影，其排名会在 *Space Jam* 的前面。所有这些电影中都没有会打篮球的外星人。我们的用户仍然对搜索结果感到失望。这样的结果，也可能会让用户对这个搜索应用感到更加不满。我们需要找到一种方法，对相关性排名进行分析，以便能够更准确地反映用户的信息需求。我们需要让 Elasticsearch 自己给出解释。下面你将会看到，对排名过程进行调试有助于我们理解：

- 如何计算每一个匹配的评价值。
- 作为影响最终结果的因素，这些匹配的评价值是如何影响文档的整体相关性评价的。

在第2章中我们已经知道了，每一个因素背后都代表着一种评价。评价就是搜索引擎给匹配查询条件的文档赋予的一个数字。它代表了文档与搜索的相关程度（评价越高意味着相关性越大）。所以，相关性排名，通常就是指对这一数字的排序。通过对排名的调试，我们会看到，尽管评价要遵循理论基础，但它完全可以按我们的意愿去操作和实现。

对匹配而言，我们必须确定，匹配的评价值是否准确反映了我们对相应特征在强度上的直觉。我们曾经说过，所有涉及外星人或与外星人相关的文本（例如，*Aliens* 或 *extraterrestrial*），其权重都应增加，以体现我们所理解的隐藏于文中的“alien-ness”特征。我们是否觉得在匹配 *alien* 时，对单词 *alien* 的评价真实地反映了我们对“alien-ness”这一文本特征在强度上的直觉呢？我们会对那些用于单词评价的数学公式进行剖析。只有这样，才能搞清楚对 *alien* 或 *basketball* 的匹配，是否真实地反映了我们的理解，即：影片真的反映了“alien-ness”或“basketball-ness”这一特征的真实强度。

我们还将看到，另一些查询通过放大处理、求和运算，以及在不同字段评价间的取舍，将多个匹配项纳入更大范围的评价计算之中，我们将了解其背后的机理。如果对词的匹配能够正确代表文本中每个特征的强度，那么此处提到的这些操作就

会把各个特征关联起来。我们的例子选择的是一个使用默认设置的 `multi_match` 查询，对片名的搜索采用了 10 倍放大，而对影片简介的搜索则没有放大。那么它是如何被翻译成一个评价公式的呢？更重要的是，如何知道根据查询条件翻译得到的这一公式是否就是我们想要的呢？

为了修正对 *Space Jam* 的查询，我们需要进入搜索引擎的“大脑”。我们需要对机械呆板的评价流程进行调整，以同时反映我们从商业和用户两个角度对相关性的理解。这包含两方面内容：如何让词与特征相关联，以及如何将这些特征的强度结合起来形成一个更高的相关性评价。

### 3.6.1 利用 Lucene 的解释功能来剖析相关性评价

Lucene 的解释 (`explain`) 功能让我们得以对隐藏于相关性评价背后的计算方法做出剖析。在深入介绍之前，让我们重温一下之前用于理解针对 *Space Jam* 的查询时所采用的方法。查询验证服务的输出结果帮助我们揭示了系统所使用的评价策略：

```
((title:basketbal title:cartoon title:alien)^10.0) |  
(overview:basketbal overview:cartoon overview:alien)
```

在这一查询中，我们想从每个字段中找出单词 `basketbal` (`basketball` 的词根)、`cartoon` 和 `alien`。我们把对片名的评价价值放大了 10 倍。随后，搜索引擎在两个字段中做出选择，取字段评价价值最大者（以 `|` 符号表示）。

我们可以从这一点出发，但是我们想看的不是策略，而是执行操作后的报告。我们想了解的是针对特定文档的评价算法。

有若干种方法都可以用来请求解释信息，但是因为我们通常会想对照每条查询的搜索结果来查看解释信息，因此在发起搜索查询的请求时，设置 `explain: true`，这种方法是最方便的。这样做会在每一条返回的搜索结果中带上一段 `_explanation` 信息。让我们带着 `explain` 的设置，重新发送一遍我们的搜索请求，以对评价过程进行考察，参见清单 3.12。

**对 Solr 读者的建议** Solr 中的参数 `&debugQuery=true` 和 Elasticsearch 中的 `'explain':True` 一样，都是用来输出同样的评价调试信息的。请见 Solr 返回的响应结果中的“`explain`”部分。

清单 3.12 请求一个相关性评价的解释信息

```

query = {
    "explain": True,
    "query": {
        "multi_match": {
            "query": usersSearch,
            "fields": ["title^10", "overview"]
        }}
}
httpResp = requests.get('http://localhost:9200/tmdb/movie/_search',
                        data=json.dumps(query))
jsonResp = json.loads(httpResp.text)
print "Explain for %s" % jsonResp['hits']['hits'][0]['_source']['title']
print json.dumps(jsonResp['hits']['hits'][0]['_explanation'], indent=True)

```

和之前一样的搜索

启用Elasticsearch的解释功能

用户的查询

从第一批搜索结果中获取\_explanation信息

完整的解释信息比较长，因此我们省略了很大一部分 JSON 输出。此处只显示了一部分。接下来我们会采用一种更为简洁的形式来显示完整的解释信息。闲话少说，下面就是针对 *Alien* 的一段 JSON 格式的解释信息。

```

{
  "description": "max of:",
  "value": 1.0643067,
  "details": [
    {
      "description": "product of:",
      "value": 1.0643067,
      "details": [
        {
          "description": "sum of:",
          "value": 3.19292,
          "details": [
            {
              "description": "weight(title:alien in 223)
                [PerFieldSimilarity], result of:",
              "value": 3.19292,
              "details": [
                {
                  "description": "score(doc=223,freq=1.0 = termFreq=1.0\n),
                    product of:",
                  "value": 3.19292,
                  "details": [
                    {
                      "description": "queryWeight, product of:",
                      "value": 0.4793294,
                      "details": [
                        {
                          "description": "idf(docFreq=9, maxDocs=2875)",
                          "value": 6.661223

```



```

    }
<omitted>
}

```

从现在开始,我们将以更为简明扼要的方式来概括这一解释格式。通过将内容压缩的方式,我们可以对上述内容进行简化,得到一份更为简短的摘要,如清单 3.13 所示。尽管简洁了不少,但内容仍旧很多。现在不用把重点放在对这些信息的理解上,只要快速浏览就可以了。我们很快就会介绍一种方法,能让大家读懂这些“天书”。

### 清单 3.13 针对 *Alien* 的简化后的解释信息

```

1.0646985, max of:
  1.0646985, product of:
    3.1940954, sum of:
      3.1940954, weight(title:alien in 223) [PerFieldSimilarity], result of:
        3.1940954, score(doc=223,freq=1.0 = termFreq=1.0
), product of:
  0.4793558, queryWeight, product of:
    6.6633077, idf(docFreq=9, maxDocs=2881)
    0.07193962, queryNorm
  6.6633077, fieldWeight in 223, product of:
    1.0, tf(freq=1.0), with freq of:
      1.0, termFreq=1.0
    6.6633077, idf(docFreq=9, maxDocs=2881)
    1.0, fieldNorm(doc=223)
0.33333334, coord(1/3)
0.053043984, product of:
  0.15913194, sum of:
    0.15913194, weight(overview:alien in 223)
      [PerFieldSimilarity], result of:
        0.15913194, score(doc=223,freq=1.0 = termFreq=1.0
), product of:
  0.033834733, queryWeight, product of:
    4.7032127, idf(docFreq=70, maxDocs=2881)
    0.0071939616, queryNorm
  4.7032127, fieldWeight in 223, product of:
    1.0, tf(freq=1.0), with freq of:
      1.0, termFreq=1.0
    4.7032127, idf(docFreq=70, maxDocs=2881)
    1.0, fieldNorm(doc=223)
0.33333334, coord(1/3)

```

下面我们比较一下针对 *Alien* 和针对目标结果 *Space Jam* 的解释信息:

```

0.08334568, max of:
  0.08334568, product of:
    0.12501852, sum of:

```

```

0.08526054, weight(overview:basketbal in 1289)
0.08526054, score(doc=1289, freq=1.0 = termFreq=1.0
), product of:
    0.049538642, queryWeight, product of:
        6.8843665, idf(docFreq=7, maxDocs=2875)
        0.0071958173, queryNorm
    1.7210916, fieldWeight in 1289, product of:
        1.0, tf(freq=1.0), with freq of:
            1.0, termFreq=1.0
        6.8843665, idf(docFreq=7, maxDocs=2875)
        0.25, fieldNorm(doc=1289)
0.03975798, weight(overview:alien in 1289)
    [PerFieldSimilarity], result of:
    0.03975798, score(doc=1289, freq=1.0 = termFreq=1.0
), product of:
    0.03382846, queryWeight, product of:
        4.701128, idf(docFreq=70, maxDocs=2875)
        0.0071958173, queryNorm
    1.175282, fieldWeight in 1289, product of:
        1.0, tf(freq=1.0), with freq of:
            1.0, termFreq=1.0
        4.701128, idf(docFreq=70, maxDocs=2875)
        0.25, fieldNorm(doc=1289)
0.6666667, coord(2/3)

```

乍一看，这些解释信息显得有些吓人。首先我们要明白的是，解释信息不过是对隐藏在相关性评价背后的算法的一个详细呈现。每一个外层的数字都是由内层嵌套的详细信息来解释的。位于最外层的解释，就是该文档的相关性评价。随着我们一层层地往下看，就可以知道这个分值是如何被逐层计算出来的。

最终，我们会到达某一层，在该层列出了针对特定匹配（如 title:alien）的评价。这一层再往下，搜索引擎就每一个匹配在各字段中的评价，对影响该评价的因素做了描述。这一层有点像解释信息中的一条分割线。在该层的内部，对每一个匹配的评价，都是通过直接访问 Lucene 的数据结构得到的，此处的数据结构指的是 Lucene 专为字段中的各个词而建的数据结构。在该层的外部，针对这些匹配的评价被组合在一起，形成了一个更大范围的公式。我们可将这一输出结果与第 2 章临近结尾处的图 2.10 所示的内容进行比较。

如果省略掉解释信息中针对每一个匹配的内部详情（也就是只看这些匹配的外层），我们就会得到一个更加简洁的针对 Alien 的解释信息：

```

1.0643067, max of:
  1.0643067, product of:
    3.19292, sum of:
      3.19292, weight(title:alien in 223) [PerFieldSimilarity]
      0.33333334, coord(1/3)
0.066263296, product of:
  0.19878988, sum of:
    0.19878988, weight(overview:alien in 223) [PerFieldSimilarity]

```

这样一来，剩下的就是对这些匹配本身的一系列操作了。从内部看，这些操作是对查询的反映，而这些查询又包含了其他查询。这种包含其他查询的查询也被称为复合查询（compound queries）。对于在底层以词匹配（term-match）的评价形式来表现的各种不同的特征，复合查询允许我们从数学的角度来表达这些特征彼此间的关联。其所体现的正是我们之前已经看到过的查询策略：

```

((title:basketbal title:cartoon title:aliens)^10.0) |
(overview:basketbal overview:cartoon overview:aliens)

```

将这些匹配组合在一起之后，它们还可以相应地被其他的复合查询在任意层级组合使用，从而创建出更为复杂的查询评价和匹配。很多相关性技术工程师都在研究，如何将一条 Query DSL 查询映射成一组复合查询。

当我们揭开面纱，仔细查看一个匹配的内部详情时，就会看到一种截然不同的计算。对评价的计算，充斥着搜索引擎的那些深奥的术语，令其显得愈发神秘莫测。此时此刻，我们所见到的，是固化于搜索引擎内部的，信息检索智能算法的一种更为本质的体现。从这一层开始，我们看到了字段匹配的统计信息。这些匹配都是评价计算的基本构成部分——但愿它们能够准确地反映文本中某个潜在特征所具有**的强度**。

```

0.03975798, weight(overview:alien in 1289) [PerFieldSimilarity], result of:
  0.03975798, score(doc=1289,freq=1.0 = termFreq=1.0
), product of:
  0.03382846, queryWeight, product of:
    4.701128, idf(docFreq=70, maxDocs=2875)
    0.0071958173, queryNorm
  1.175282, fieldWeight in 1289, product of:
    1.0, tf(freq=1.0), with freq of:
      1.0, termFreq=1.0
    4.701128, idf(docFreq=70, maxDocs=2875)
    0.25, fieldNorm(doc=1289)

```

同样的，大家可能又会被吓到！别担心。后面我们会介绍这些数字背后的原理，

到那时，我们就可以轻松地对这些匹配加以对比，能够确定为什么有些字段的匹配看起来会比其他字段的在强度上显得更具优势。

### 3.6.2 向量空间模型、相关性解释信息和我们

Lucene 的许多评价公式都来源于信息检索。然而理论对公式的影响却需要进行大幅度的调整。尽管理论基础可以为我们解决问题提供帮助，但在现实中，相关性评价所使用的，则通常是从理论出发、在实践应用中得到验证的方法。除了那些基本的概念，相关性评价在很多方面既是一门科学也是一门艺术。理解了它，就可以帮助我们确保搜索引擎能够对潜藏于文本中、以单词形式表现出来的特征，在权重上做出正确的度量。

对于信息检索而言，在一个字段中对多个词的搜索（比如我们对 *Space Jam* 的搜索：overview:basketbal、overview:alien 和 overview:cartoon）大体上就是查询条件和匹配文档之间的一个向量比较。向量？听起来就像是在用几何学来解决“语言艺术的问题（a language arts problem）”。回忆一下，向量表示空间中一个具有大小（magnitude）和方向（direction）的量。向量通常是用一个从原点出发指向空间的箭头来表示的——比如从地球到月球。如果用数字来表示，一个向量可以用每个维度上的一个数值来表示。也许向量  $\langle 50, 20 \rangle$  表示“北 50 英里，东 20 英里”。不过，对于向量而言，其空间并不一定就是我们所处的物理世界。例如，如果  $y$  轴表示一个水果的果汁含量， $x$  轴表示它们的尺寸大小，我们就可以定义一个向量空间，来反映水果的某些重要特征。图 3.2 所展示的就是这样一个向量空间。

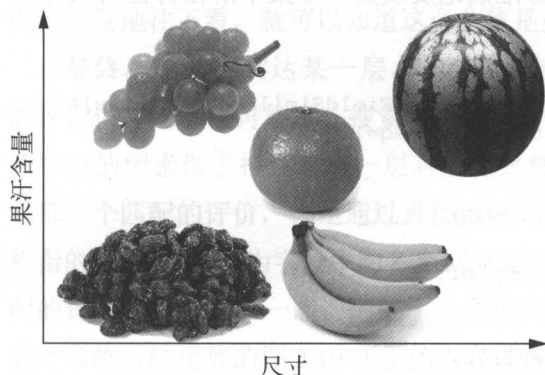


图 3.2 在一个二维向量空间中的水果； $x$  轴代表尺寸大小， $y$  轴代表果汁含量。每一种水果，按照尺寸大小/果汁含量进行度量，都可以用一个向量来表示，度量结果相近的水果分布在一起。

在这张图上，我们可以看到在尺寸大小 / 果汁含量这一向量空间中用向量表示的几种水果。其中一些水果在果汁含量的维度上具有较大的强度，而其他水果则在尺寸大小的维度上具有较大的强度。我们可以很容易地看到，相近的水果可能会重叠在一起。例如，位于右上角的，最有可能是西瓜一类的水果——因为它们又大又多汁。

我们可以通过计算两个向量的点积来推断出两种水果的相似度。在这个水果的例子中就是，(1) 将每种水果的果汁含量相乘，(2) 将每种水果的尺寸大小相乘，(3) 两个结果相加。我们可以相应得出这样的结论：水果的共性越多，点积的取值越大。

```
dotprod(fruit1, fruit2) = juiciness(fruit1) × juiciness(fruit2) +  
                           size(fruit1) × size(fruit2)
```

这和文本又有什么关系呢？对于信息检索而言，文本（查询条件和文档）同样也可以用向量来表示。我们不再使用果汁含量和尺寸大小这样的特征了，在文本的向量空间里，维度表示的是单词在文本中出现的可能性。如果我们讨论的不是水果，而是影片简介中是否提到了 basketball、aliens 或 cartoons，如图 3.3 所示，情况又该如何呢？一些文本中明确涉及了 aliens（例如，电影 *Alien* 的简介），但却并没有涉及 basketball 和 cartoons。而其他文本（如日本动画电影 *Slam Dunk*，中文名为《灌篮高手》）的简介则是关于 basketball 和 cartoons 的，但它们却和 aliens 无关。我们觉得我们的搜索目标 *Space Jam* 在所有要求的维度上应该都有很高的评价价值才对。

水果有果汁含量这样的特征，同样的，根据单词 alien 在文本中出现的情况，我们可以认为影片文本具有“alien-ness”的特征。总结一下这种特征的表现方式：我们可以定义一个特征空间来代表向量空间，用维度来代表特征，不管我们所讨论的特征具体是什么，水果、文本，或任何其他值得比较的东西都可以。

当然，电影远不止篮球、卡通和外星人这几个主题。对于文本来说，特征空间不止三个维度。在文本的所谓词袋模型（bag of words model）中，我们的向量针对每一个可能的词语都有一个单独的维度。在英语中，每个单词都有可能是一个维度！当然，任何给定的文档或查询都不太可能提到英语中的每一个单词。例如我们很难在 *Space Jam* 的简介中找到 Rome 或 history 这样的词。类似的，*Gladiator*（中文名《角斗士》）里也不可能提到 Michael Jordan。因此，在这些文档向量中，大部分维度都是空或者零。正是由于这个原因，它们也被称为稀疏向量（sparse vector）。



图 3.3 影片简介的文本在一个由三个维度构成的向量空间中, 这三个维度分别是 basketball、cartoon 和 alien。一些电影与 cartoons 和 basketball 非常相关 (如 *Slam Dunk*), 但 *Space Jam* 与这三个都非常相关!

在了解了每个向量的维度都是一个特征之后, 下一步就是度量这一特征的大小或强弱了。在搜索中, 这个数值被称作权重 (weight) ——即某个词针对一段文本在重要程度上的一种度量。例如, 如果 alien 在文本中的地位很显著, 那它就应该得到较高的权重; 否则, 就应该得到较低的权重, 或者权重为零。如果我们重新审视之前的解释信息, 就会看到 Lucene 对 *Space Jam* 中 “alien” 的维度所给出的权重度量结果为:

```
0.03975798, weight(overview:alien in 1289)
```

在进一步探究 Lucene 的权重计算方法之前, 让我们先来看一个更为简单的例子。对于文本中的某个单词, 如果它出现在文本中, 我们定义其权重为 1, 反之定义为 0。有了这样的一个定义, 一段取自影片 *Space Jam* 简介中的文本——“basketball game against alien”, 以词袋向量  $v_o$  的形式来表示就是:



a	alien	against	...	basketball	Cartoon	...	game	...	movie	narnia	...	zoo
0	1	1		1	0	0	1		0	0		0

该向量针对英语中的每一个单词都有一个维度；我们在这里显示的仅仅是少数几个英文单词。我们可以将其和另一个以同样方式构建的向量进行比较，即针对我们的查询条件“basketball with cartoon aliens”而构建的向量  $V_Q$ ：

a	alien	against	...	basketball	Cartoon	...	game	...	movie	narnia	...	zoo
0	1	10		1	1	0	0		0	0		0

这两个向量中有几处是匹配的？查询和文档有多相似？就像那个水果的例子一样，我们可以通过计算点积来得到评价值。回忆一下，点积就是两个向量中对应元素两两相乘后再求和的结果。因此，我们对查询条件的评价，其公式计算如下所示：

$$\text{score} = V_D['a'] \times V_Q['a'] + V_D['alien'] \times V_Q['alien'] + \dots + V_D['space'] \times V_Q['space'] \dots$$

对比之前解释信息的详情，每一个乘积都代表了一个匹配评价值。也就是说，在解释信息中，overview:alien 对应的是  $V_D['alien'] \times V_Q['alien']$ 。有所不同的是，解释信息中所反映的是 Lucene 自带的函数，用来计算字段或查询的权重，接下来我们会深入研究。前面对点积的求和运算从这里布尔查询的行为中也可以找到，它会把所有匹配的子句累加求和。我们可以从之前解释信息中的 sum of 处看到这一求和运算：

```
3.19292, sum of:
  3.19292, weight(title:alien in 223) [PerFieldSimilarity]
```

### 3.6.3 向量空间模型在实践中的注意事项

尽管向量空间模型为我们讨论 Lucene 的评价提供了一个通用的框架，但它还远不是一个完整的视图。在实践中，许多经验系数（fudge factor）都已经被证明可以改进评价的分值。或许最根本的一点是，通过将查询组合起来得到更大粒度的评价，这种组合匹配项的方式并不一定要利用相加求和来完成。

我们已经知道了，通常符号 | 表示取两个字段中的“最大值”。如果组合的匹配项缺少某些部分，通常还有一个放大因子（coord 值）可以直接对此进行“评判”（点积的结果乘以 coord 值，即： $< \text{匹配数} > / < \text{查询词总数} >$ ）。我们即将遇到的许多复合查询，都会在底层查询的基础上执行各种操作，比如取最大值、相加，或相乘。

我们还拥有极大的自由度，可以利用自己定义的函数查询（function queries）对评价进行任意形式的计算或放大处理，此处的函数查询可以将针对匹配（或其他内容）的评价，与任何其他形式的因子相结合，再组合到一起。我们会在后面的章节中探讨许多这样的策略。

对于上述点积运算的另一个要点是，通过除以每个向量的长度，我们通常会对其进行归一化处理（normalize）：

$$\text{score} = \frac{(V_D['a'] \times V_Q['a'] + V_D['alien'] \times V_Q['alien'] + \dots + V_D['space'] \times V_Q['space'])}{(|V_Q| \times |V_D|)}$$

对于点积而言，归一化处理把评价转换成了一个介于 0 到 1 的数值。这样就可以让公式重新得到平衡，对有较高权值的特征和权值相对较低的特征进行考量。<sup>1</sup> 对于搜索来说，考虑到 Lucene 的评价计算中出现的各种经验系数，以及千奇百怪的字段统计方法，为了让不同查询的评价具有可比性，如果没有做过大量深入的定制，最好不要尝试对两个查询条件的评价值进行比较。

正如之前所说，文本以稀疏向量的形式表现，也被称作“词袋”模型。它之所以被称作“袋（bag）”，是因为其所体现的对文本的分解，忽略了词汇的上下文。这种上下文的一个重要部分就是词语在文中出现的位置，它是用来进行短语匹配的。幸运的是，Lucene 也存储了每个词语出现的位置。因此，我们可以把一篇文档看成是一个包含了所有子短语（subphrase）的稀疏向量。这的确会是一个非常巨大的向量！以下表为例。打开复杂的 span queries 库，你会发现这种向量的规模甚至还要更大！

a	an	alien	...	Basketball	lump	...	"basketball game"	"game against"	...
0	0	1		0	0		1	1	

### 3.6.4 通过对匹配的评价来度量相关性

我们仍然还在寻根溯源，为什么有些影片会匪夷所思地排在我们的目标影片 *Space Jam* 的前面。我们已经看过解释信息，找到了一些值得注意的匹配评价。马上就要胜利啦！

我们需要理解，Lucene 在其向量计算的过程中，是如何对一篇文档或查询文本

<sup>1</sup> 精明的读者会发现，这就是余弦相似度（cosine similarity）。

中的词语进行权重度量的。然后我们就可以根据匹配应被赋予的权重大小，对这些权重是否符合我们的直觉做出评估。用户在搜索时当然不考虑此处所提到的这些数学运算。但这些度量方法已经得到实践证明，在广义上是接近用户对相关性的感受的。让我们来看一下在这个例子中它是否与我们的预期相符。

一起来看一看 Lucene 针对 alien 在 *Space Jam* 中的权重计算，弄清楚为什么针对 alien 的匹配，其评价价值相对较低：

```
0.03975798, weight(overview:alien in 1289) [PerFieldSimilarity], result of:
  0.03975798, score(doc=1289,freq=1.0 = termFreq=1.0), product of:
    0.03382846, queryWeight, product of:
      4.701128, idf(docFreq=70, maxDocs=2875)
      0.0071958173, queryNorm
    1.175282, fieldWeight in 1289, product of:
      1.0, tf(freq=1.0), with freq of:
        1.0, termFreq=1.0
      4.701128, idf(docFreq=70, maxDocs=2875)
      0.25, fieldNorm(doc=1289)
```

Lucene 的权重计算是如何做的呢？看起来像是两个部分的权重值相乘。fieldWeight 所反映的，是 Lucene 对该词在字段（此处为 overview）文本中重要性的计算。而 queryWeight 则用于该词在用户查询中的权重计算。

针对查询和正在接受评价的两篇文档，我们可以根据解释信息将其权重信息翻译成稀疏向量（也就是前面部分所说的  $V_q$  和  $V_d$ ）。例如，如果我们把 alien 在 *Space Jam* 中的权重和它在 *Alien* 中的权重进行比较：

```
0.15913194, weight(overview:alien in 223) [PerFieldSimilarity], result of:
  0.15913194, score(doc=223,freq=1.0 = termFreq=1.0), product of:
    0.033834733, queryWeight, product of:
      4.7032127, idf(docFreq=70, maxDocs=2881)
      0.0071939616, queryNorm
    4.7032127, fieldWeight in 223, product of:
      1.0, tf(freq=1.0), with freq of:
        1.0, termFreq=1.0
      4.7032127, idf(docFreq=70, maxDocs=2881)
      1.0, fieldNorm(doc=223)
```

我们可以用一个稀疏向量来代表这些权重。下面是 alien 的权重：

查询或字段	...	alien	...
查询：basketball with cartoon aliens ( $V_q$ )		0.033834733	
<i>Space Jam</i> 中的 overview 字段 ( $V_d$ )		1.175282	
<i>Alien</i> 中的 overview 字段 ( $V_d$ )		4.7032127	

由于某些原因，单词 alien 在 *Alien* 的 overview 字段中的权重比它在 *Space Jam* 中的更高。对我们而言，这意味着特征“alien-ness”在 overview 文本中的评价更高。

### 3.6.5 用 $TF \times IDF$ 计算权重

计算词语在字段中的权重，其规则是由 Lucene 中所谓的相似度（similarity）来决定的。相似度利用保存在索引中的、获得匹配的单词的统计信息，帮助查询条件为该词求得一个权重值。Lucene 支持若干种相似度的实现，也包括让我们定义自己的实现。

大部分相似度的计算都是基于公式  $TF \times IDF$  的。这里指两个重要的词统计信息的乘积，它们通过字段提取得到，并被 Lucene 记录在倒排索引中——即词频（TF）和逆文档频率（inverse document frequency，即 IDF）。默认情况下，词的重要性就是通过这两个统计值的乘积计算出来的。也许大家还记得，我们在第 2 章的末尾讨论过这些统计值。让我们重新回顾一下。

TF（即前述评价计算中的 *tf*）表示词语在字段中出现的频率。我们可以在前述 SimpleText 版的倒排索引中看到这一统计值，即 *freq*。TF 在对匹配进行评价时非常有价值。如果一个匹配词在一篇文档的某个字段中频繁出现（比如字段中多次提到 alien），我们就认为这一字段的文本很有可能是涉及该词的（在本例中，我们认为它很可能是关于外星人的）。

相反的，IDF（即前述评价计算中的 *idf*）告诉我们匹配词的罕见程度（因此它是有价值的）。因为 IDF 是文档频率的倒数，由  $1 / \text{文档频率}$ ，或  $1 / DF$  计算得到。也许大家还记得，DF 记录的是出现某个单词的文档数量。如果该词很常见，文档频率就会很高。稀有的词被认为更有价值，而常见的词则相反。如果单词 *super-califragilistic* 只在一篇文档中出现，那它的 IDF 值就会很高。

未经修饰的  $TF \times IDF$  是通过将 TF 和 IDF 相乘来为文中单词的重要性赋予权重的——也可以写成  $TF \times (1 / DF)$ ，或  $TF / DF$ ，以此来度量该词在索引中的全部使用中有多少比例是集中在某篇文档中的。

表 3.1 展示了  $TF \times IDF$  的工作方式。在本例中，当我们计算 lego 的重要性时，有关乐高积木的影片相对而言非常少见。正如我们预期的那样，一部涉及乐高的电影，*The Lego Movie*（中文名《乐高大电影》），得到了较高的  $TF \times IDF$  权重

值。将针对 lego 的搜索和针对 love 的搜索进行对照，涉及单词 love 的电影特别常见（每一个人都喜欢浪漫喜剧）。这就导致了，love 在浪漫喜剧影片 *Sleepless in Seattle*（中文名《西雅图夜未眠》）中的出现，所得到的权重要低于 *The Lego Movie* 中的 lego，尽管 love 在 *Sleepless in Seattle* 中出现的次数多了很多。

表 3.1 love 匹配 *Sleepless in Seattle* 与 lego 匹配 *The Lego Movie* 的评价值对比。lego 很少见且只在 *The Lego Movie* 中被提到，因此相比于 love 的匹配获得了更高的评价值

影片	匹配词	DF	TF	$TF \times IDF (TF / DF)$
<i>Sleepless In Seattle</i>	love	100	10	$10 / 100 = 0.1$
<i>The Lego Movie</i>	lego	1	3	$3 / 1 = 3.0$

文本中什么样的单词匹配才应该被认为是更重要的，对于这一点， $TF \times IDF$  背后的思想是符合大部分用户的直觉的。用户认为稀有词（如 lego）要比常见词（如 love）更特殊也更有针对性。而且，当一段文本提及某个单词的比例大于其他文本时（也就是 TF 值增加），那么很有可能这段文本就是在讨论我们正在搜索的那个单词。

尽管这种方法具有广泛的价值，我们还是会遇到一些违背直觉的情况。有时，增加 TF 的值与用户对单词重要性的认识并不相关。那些匹配小段文本的高 TF 值，例如，片名字段（如 *Fire with Fire*）通常就和我们对单词权重增加的认识是不相关的。所幸的是，Elasticsearch 允许我们根据需要禁用 TF。

### 3.6.6 谎言、该死的谎言和相似度<sup>1</sup>

虽然我们可以看到， $TF \times IDF$  貌似是一个很直观的权重计算公式，但这些未经处理的统计数据仍然需要额外的调节才能达到最优。对信息检索的研究证明，虽然一个搜索词在一段文本中可能出现 10 次以上，但这并不能说明它具有 10 倍的相关性。单词被提到的次数越多确实会与相关性有关系，但这种关系并不是线性的。正是由于这个原因，Lucene 通过相似度类（similarity class）对 TF 和 IDF 的影响进行了抑制。

本书使用了 Lucene 经典的  $TF \times IDF$  相似度（在 Solr 5.x 中和 Elasticsearch 2.0 中是默认的——见后面的“Lucene 的下一代默认相似度：BM25”补充资料）。它在计算权重时对 tf 和 idf 的影响进行了抑制。

<sup>1</sup> 此处借用了成语“Lies, damned lies, and statistics”，意指单纯的统计数据并不总是能够准确地反映用户的预期，具体信息可参见[https://en.wikipedia.org/wiki/Lies,\\_damned\\_lies,\\_and\\_statistics](https://en.wikipedia.org/wiki/Lies,_damned_lies,_and_statistics)。——译者注



TF Weight =  $\sqrt{\text{tf}}$

IDF Weight =  $\log(\text{numDocs} / (\text{df} + 1)) + 1$

我们可以从表 3.2 中看到这些统计信息是如何被抑制的。

表 3.2 利用默认公式对词频和文档频率进行抑制 (IDF 是根据 1000 个文档计算得到的)

原始的 TF	TF 的权重	原始的 DF	IDF 的权重 (针对 1000 个文档)
1	1	1	7.215
2	1.414	2	6.809
5	2.236	10	5.510
15	3.872	50	2.976
50	7.071	1000	0.999
1,000	31.623		

还有, 只对  $\text{TF} \times \text{IDF}$  自身进行抑制往往是不够的。我们通常认为词频应当是相对于匹配字段中的单词总量而言的。例如, alien 在一本 1000 页的书中出现一次和它在一段只有三句话的文本中出现一次会有相同的权重吗? 单词在篇幅较短的文本中只出现一次也许要比在一本书中只出现一次更具相关性。正是由于这个原因, 在处理 fieldWeight 的计算时, 我们将  $\text{TF} \times \text{IDF}$  乘上了 fieldNorm——它是一个基于文档长度的权重的归一化因子 (weight-normalization factor)。该归一化因子是通过下面的公式计算得到的:

$\text{fieldNorm} = 1 / \sqrt{\text{fieldlength}}$

归一化处理更倾向于短文本字段中单词的出现, 以此校正 TF 和 IDF 对单词权重的影响。归一化是在索引阶段计算的, 并且占用了磁盘空间。而且, 它还跟应用类型和用户需求有关, 并不总是和用户所认为的, 单词在一段文本中的重要性, 保持相关性。所幸的是, Lucene 允许我们完全禁用归一化。

综上, Lucene 经典的相似度是按照如下公式来度量一段文本中的单词权重的:

$\text{TF weighted} \times \text{IDF weighted} \times \text{fieldNorm}$

重新回顾一下 fieldWeight 的计算, 我们就明白了这一公式在起着作用:

0.4414702, fieldWeight in 31, product of:  
 1.4142135, tf(freq=2.0), with freq of:  
 2.0, termFreq=2.0  
 3.9957323, idf(docFreq=1, maxDocs=40)  
 0.078125, fieldNorm(doc=31)



## Lucene 的下一代默认相似度：BM25

近几年，另一种计算  $TF \times IDF$  评价的方法在信息检索社区中变得十分流行：它就是 Okapi BM25。由于它在处理长文本（article-length text）时被证明具有很高的性能，所以 Lucene 将会推出 BM25，作为 Solr 和 Elasticsearch 的默认相似度，甚至当大家读本书时，也许它已经就是默认相似度了。

什么是 BM25 呢？和我们之前所讨论的“经验系数”不同，BM25 在  $TF \times IDF$  方面的“经验”是以信息检索领域中更为可靠的科学发现为基础的。这其中包括，能够让  $TF$  所产生的影响受到一个饱和值（saturation point）的约束。不同于让长度带来的影响（fieldNorms）无限制地增长，BM25 对这种影响的计算是相对于平均文档长度而言的（超过平均长度的文档，权重会下降，而低于平均长度的文档，则权重会增加）。对于  $IDF$  的计算则和经典的  $TF \times IDF$  相似度差不多。

BM25 会对我们的相关性计算有帮助吗？答案并不是那么简单。正如我们在第 1 章中讨论过的那样，信息检索重点关注的是对长文本广泛而持续的改进。而针对特定应用的相关性定义，BM25 或许并没有多大作用。正是由于这个原因，我们不打算在本书中再对 BM25 做深入介绍。Lucene 不会完全抛弃经典的  $TF \times IDF$ ，而是把它变成所谓的经典相似度（classic similarity）。这两种方法我们都试一试。至于本书中的那些例子，我们在使用后续的 Elasticsearch 版本时，可以把相似度改回经典相似度，重新计算评价值。最后，无论选择 BM25 还是经典的  $TF \times IDF$ ，我们从本书中学到的每一课都是有用的。

### 3.6.7 决定搜索词重要性的因素

我们对查询条件的权重计算（queryWeight）使用的并不是同一个公式。在几乎所有的搜索案例中，一个单词在查询条件中出现的次数越多并不代表它越重要（基本上用户的每一个查询词都只会列出来一次）。而且，因为查询的长度比较短，所以基本不需要做长度归一化处理。因此这一步也可以被省略掉。我们之前介绍的权重计算公式就只剩下  $IDF$  了。

另外，queryWeight 还增加了两个因子：

- 查询期放大（query-time boosting）

#### ■ 查询归一化 (query normalization, 即 queryNorm)

首先,我们将避开 queryNorm 的讨论。第一点要说明的是,如果不进行放大处理的话,queryNorm 是无关紧要的。对于一条搜索的所有匹配来说,它都是一个常量。queryNorm 试图使这条搜索的不同匹配间的评价值具有可比性,但效果并不好。所以请不要尝试针对一条搜索的不同字段,在这些字段之间进行评价比较。在统计信息中,有很多像 IDF 和 TF 这样跨越字段和文本的变量,可以让相关性的评价变得非常相关。实际上,在 Lucene 的讨论组里人们一直在考虑去掉这个因子。<sup>1</sup>

对 queryWeight 真正起作用的是放大系数 (boost factor)。在我们的查询中并没有对影片简介进行放大处理,但是我们的确对片名匹配使用了放大。不幸的是,在对 queryNorm 的计算过程中,这种放大有时可能会丢失。查看 queryNorm 的计算我们会发现,在影片 *Alien* 中,针对片名匹配的 queryNorm,其计算方法不同于简介字段中的相应匹配,它使用了一个 10 倍的放大系数。在后续章节中大家会看到,Elasticsearch 允许我们对不同的因素进行放大处理。

### 3.6.8 解决 Space Jam 和 alien 的排名问题

终于掌握了 Lucene 的评价计算,我们可以对 *Space Jam* 和 *Alien* 的解释信息进行比较了。*Alien* 有两处匹配:一个是针对片名的强匹配,一个是在简介字段中相对较弱的匹配。而 *Space Jam* 则在简介字段中有两个匹配。如果我们仔细研究一下导致匹配计算方法不同的原因,就会发现对简介字段的评价通常总是要比对片名字段的评价弱很多。

我们可以看到针对 *Alien* 的片名匹配有一个很高的评价:

```
3.1940954, weight(title:alien in 223)
```

对比简介字段中一个有关 alien 的匹配,其相关性评价相对较低:

```
0.03975798, weight(overview:alien in 1289)
```

两者差了大概两个数量级!等一下,我们不是通过放大处理,已经明确地告诉搜索引擎片名只比简介重要 10 倍吗?没错,虽然我们做了,也知道字段之间的评

---

<sup>1</sup> 更多细节可参见 *Whither Query Norm*, 访问地址为 <http://lucene.472066.n3.nabble.com/Whither-Query-Norm-td600443.html>。

价值是完全没有可比性的。它们完全存在于自己的评价空间中。将针对 Alien 的片名匹配和针对 alien 的简介匹配进行比较，我们可以看到：

```

3.1940954, weight(title:alien in 223) [PerFieldSimilarity], result of:
  3.1940954, score(doc=223,freq=1.0 = termFreq=1.0
), product of:
  0.4793558, queryWeight, product of:
    6.6633077, idf(docFreq=9, maxDocs=2881)
    0.07193962, queryNorm
  6.6633077, fieldWeight in 223, product of:
    1.0, tf(freq=1.0), with freq of:
      1.0, termFreq=1.0
    6.6633077, idf(docFreq=9, maxDocs=2881)
    1.0, fieldNorm(doc=223)
0.03975798, weight(overview:alien in 1289) [PerFieldSimilarity], result of:
  0.03975798, score(doc=1289,freq=1.0 = termFreq=1.0), product of:
    0.03382846, queryWeight, product of:
      4.701128, idf(docFreq=70, maxDocs=2875)
      0.0071958173, queryNorm
    1.175282, fieldWeight in 1289, product of:
      1.0, tf(freq=1.0), with freq of:
        1.0, termFreq=1.0
      4.701128, idf(docFreq=70, maxDocs=2875)
      0.25, fieldNorm(doc=1289)

```

① “alien”  
在简介字段中的IDF值相比之下低了很多，同时简介的长度也比片名更长，因此片名字段的归一化因子更大

此处的queryNorm值是前面queryNorm的1/10，反映了对片名的放大作用

此处，我们找到了导致两个匹配在针对 fieldWeight 的评价上有所不同的原因①。这些字段都有自己不同的特点。影片的简介大致是一个段落的篇幅，而片名则只有几个单词。这往往导致了两者的 fieldNorms 值截然不同。而且，单词在简介字段中的相对分布和在片名字段中的分布也是不一样的。

字段的作者选择何种方式来表达自己的想法，这一点常常对字段起决定作用。影片的营销团队在写简介的时候是如何考虑的？他们会选择用什么单词？写得简明扼要还是内容详尽？电影工作室会如何选择电影的片名？他们会沿用现有的品牌（以及那些词汇），还是始终努力保持原创？通常，文本的相关性需要同时兼顾作者（比如为什么作者会选择某种语言）和搜索者（比如为什么搜索者会使用特定的搜索词）。

对于我们要修复的计算公式来说，主要的问题是，一个针对简介的有效评价可能会比一个针对片名的有效评价在数值上小很多。对搜索引擎来说，放大 10 倍并不意味着就有 10 倍的重要性，它只代表了一个乘数而已。当我们对词匹配进行分析时，可以很明显地看到，对简介的评价在数值上总是远远低于对片名的评价。对于这些字段的权重，正确的使用方法是：在决定如何设置权重之前，要先对这些评价值的

准确性有一个初步的研究。也许我们把片名放大 0.1 倍会更合适，这样做给针对片名的匹配所带来的权重，也许依然远远大于简介，这只不过是因为片名字段所具备的一些特殊性而导致的。

让我们用一个更合理的片名放大系数重新执行一次查询，看看效果如何，代码如下清单 3.14 所示。

清单 3.14 利用调整后的放大系数进行搜索

```
query = {  
  "query": {  
    "multi_match": {  
      "query": usersSearch,  
      "fields": ["title^0.1", "overview"],  
    }  
  }  
}  
search(query)
```

用户的查询

结果：

Num	Relevance Score	Movie Title
1	1.0016364	Space Jam
2	0.29594672	Grown Ups
3	0.28491083	Speed Racer
4	0.28491083	The Flintstones
5	0.2536686	White Men Can't Jump
6	0.2536686	Coach Carter
7	0.21968345	Semi-Pro
8	0.20324169	The Thing
9	0.1724563	Meet Dave
10	0.16911241	Teen Wolf

太好了！现在看起来好多了。

### 3.7 问题解决了？工作永远做不完！

我们取得了一些进展，工作有所推进，但是真的解决所有问题了吗？还是有一些地方需要改进。

首先，根据之前验证服务的返回结果，回忆一下我们的查询是如何工作的：

```
((title:basketbal title:cartoon title:alien)^10.0) | (overview:basketbal  
overview:cartoon overview:alien)
```

还记得符号 | 是如何从两个字段的的评价中取最大值的吗？在下面的解释信息①

中也可以看出来，我们只讨论复合查询部分。

```
1.0643067, max of:
```

```
1.0643067, product of:
```

```
3.19292, sum of:
```

```
3.19292, weight(title:alien in 223) [PerFieldSimilarity], result of:
```

```
0.33333334, coord(1/3)
```

```
0.066263296, product of:
```

```
0.19878988, sum of:
```

```
0.19878988, weight(overview:alien in 223) [PerFieldSimilarity], result of:
```

为什么取两个评价中的最大值呢？这样做的目的是什么呢？

通过对简介和片名更为严谨地放大处理，当查询语句要从两个字段中取最大值时，我们可以做到让简介的权重有可能胜过片名。为什么我们要取最大值呢？为什么要默认使用这样的策略呢？有没有其他策略可以把这些评价价值综合起来，使得在同时面对片名和简介的两个强匹配时，不是二者择其一呢？或许我们已经解决了搜索 *Space Jam* 的问题，但是这种取最大值的方法给其他的搜索带来了什么呢？当其他搜索产生了不同的评价价值，我们是否还要从头再来呢？

最后，你一定会问，对于 `fieldWeight` 的计算是否还有改进的余地。我们真的很关心 `fieldNorms` 吗？在这个案例中，对短文本或长文本的倾向很重要吗？还有一个对相关性技术工程师来说一直存在的困扰：词语本身是否准确地代表了隐藏在文本中的特征呢？如果看过一小段 *Space Jam*，也许我们就会思考一系列问题：

Michael Jordan 同意帮助 Looney Toons 与外星怪物们打一场篮球赛，从而决定 Looney Toons 的去留。

所有特征在解释信息中真的都被捕获到了吗？我们在解释信息中并没有看到任何匹配 `cartoon` 的权重；`toons` 或 `looney toons` 应该匹配 `cartoon` 吗？Michael Jordan 的匹配呢？人们通常会把他和篮球联系在一起；看到他的名字出现，我们是否应该放大单词 `basketball` 的权重呢？

把潜在的特征转化成词，并加以组合，是相关性技术工程师一直以来奋斗的目标。在第 4 章，我们会详细介绍这一主题！我们已经有了一个很好的开端。在后面几章中，我们会继续深入探究这些问题，为了同时满足用户需求和业务需求，我们需要最大限度地利用搜索工具来进行匹配和排名。

## 3.8 本章小结

- 没有得到预期的搜索结果，需要我们对匹配和排名进行调试。

- 为了对匹配进行调试, 请检查搜索引擎解释和执行查询的方法; 对于 Elasticsearch 而言, 就是使用查询验证服务。
- 为了在搜索结果中包含某篇文档, 搜索引擎要求精确地、逐字节地进行词匹配。
- 搜索引擎需要我们的帮助, 通过选择合适的分析器, 确定哪些匹配应该被包含, 哪些应该被忽略(如禁用词)。
- 搜索引擎利用  $TF \times IDF$  评价算法对结果进行排名。通过搜索引擎提供的相关性解释信息的输出结果, 我们可以了解其评价的过程。
- TF 可以在文档的文本中对单词的重要性进行度量。IDF 则决定了单词在整个文档集中的稀缺性或特殊性。字段的归一化处理倾向于对短文本的评价。
- 调试搜索引擎的评价算法需要我们理解 `fieldWeight` (即搜索词在文本中的重要性), 并对 `queryWeight` 进行调试(即搜索词在搜索查询中的重要性)
- 相关性评价如果不进行归一化处理是无法轻易进行比较的。一个对片名字段较低的评价也许是 10.0, 而一个对简介字段较高的评价也许只有 0.01。
- 放大系数并非字段的优先级, 而是我们让相关性评价重新获得平衡的手段。当评价得到平衡之后, 我们可以再调整字段的优先级。



# 驾驭token

## 本章要点

- 分词是用来提取语义而非单词的
- 搜索中查准率和查全率的概念
- 在查准率和查全率之间做出权衡
- 控制匹配的专指性
- 将非文本数据编码到搜索引擎中

到这里，我们已经充分理解了为什么相关性对搜索应用的成功如此重要（参见第1章）。我们也掌握了有关搜索引擎内部机制的实用知识（参见第2章），并学会了相关性问题的调试方法，明白了文档为什么匹配，为什么这样评分（参见第3章）。

现在，有了动力、知识和工具，是时候深入探讨一下相关性工程（relevance engineering）这门技术了。本章，我们关注的是文本分析。恰当的分析是相关性搜索的基础。正如我们在第3章里看到的那样，分析控制着匹配。如果分析得当，用户的查询就只会匹配他们想找的文档。但如果分析不当，用户的查询就会匹配到许多不相关的文档，或者有可能匹配不到任何文档！

## 4.1 将token作为文档特征

我们曾多次指出相关性和分类的联系。（还记得那个水果的例子吗？）这种联系在我们讨论 token 时也许是最明显的，因为正如水果可以用它的颜色、形状和大小作为特征来分类一样，文档也可以用从中提取的 token 作为特征来分类。

让我们回到水果的例子。人们能想到的最明显的特征有颜色、形状和大小。事实上，我们还可以尝试用其他特征对水果进行分类：重量、气味、价格、斑块的数量、价签的颜色，还有到华盛顿纪念碑的距离。不过正如我们在这个列表里所看到的那样，有些特征比起其他特征来，对分类的用处更大。而有些特征，比如到华盛顿纪念碑的距离，则对分类没有丝毫用处。

同样的，我们采用文本分析来提取特征——token——对用户预期做出预测，并给出高度相关且有针对性的结果。但是如果文本分析配置不当，就可能导致无效的特征，令搜索体验多半成为浪费时间。文本分析既是一门科学（science），也是一门艺术（art）。在第2章中，我们主要讨论的是分析的原理——这是科学，如果你愿意接受这种说法的话。在这里，我们讨论的则是分析更为软性、更为“艺术（artistic）”的一面，我们称之为特征建模（feature modeling）。

借助特征建模，我们同时考虑了用户查询的意图和文档揭示的语义。还确保了由分析过程所生成的 token 不仅代表查询的特征，也代表文档的特征，让这些特征生动而有意义（如图4.1所示）。

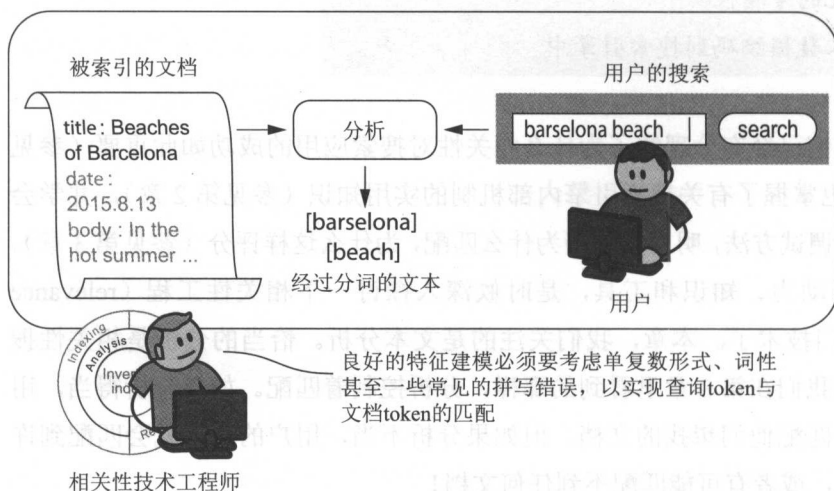


图 4.1 相关性技术工程师确保由文档生成得到的 token 与由查询生成得到的 token 相匹配。为此，他必须理解文档的信息以及用户的意图。

### 4.1.1 匹配的流程

还是拿水果来举例……假设你腹中空空，但并非饥不择食；而是很想尝一尝红蛇果（red delicious apple）。于是为了过嘴瘾，你直奔水果摊位的一角。到跟前，才发现水果种类繁多：

- 长长的黄香蕉，有的带斑，有的青绿。
- 小而圆的葡萄，有的浅绿，有的深紫。
- 切成瓣的绿皮红瓤大西瓜，有的有子，有的没子。

不过今天你是来找苹果的。在你心里，有一个清晰的模板——如果愿意，也可以称之为查询——红蛇果看上去应该是什么样的。它是一种红色的、圆圆的水果，大小和拳头差不多。另外，上好的红蛇果总是油光锃亮的！

你四处打量，一眨眼的时间，就从一堆水果里找到了和你的查询最相匹配的一个“子集”。你走到装有苹果的箱子跟前。摆在最上面的就是那油光锃亮、无比美味的红蛇果。搜索带给了我们允许范围内相关性最好的结果。你买到了苹果并狼吞虎咽地吃了起来！（直到吃了一半，才注意到苹果里还有一条虫子。也许下次你应该细化一下查询条件！）

这一场景是对用户在文档搜索时所遇情形的一个很好的类比。用户在访问搜索应用时，是有其信息需求的，他们提供给搜索引擎一组搜索词，或一个短语，这是他们认为最能描述其需求的单词或短语。相应的，搜索引擎收到这些词，并通过第2章中介绍的分析流程将其转换成 token。然后，搜索引擎快速扫描整个索引，把含有这些 token（即由查询生成的 token）的文档搜集起来，根据相关性对文档顺序加以组织，并将结果返回给用户。

### 4.1.2 token，不只是单词

此处一定要注意，token 的生成——以及相应特征的生成——对查询和文档而言都是存在的。只有当特征始终是根据查询的文档得来的，匹配才可能发生。而且，只有当特征充分捕获了文档和用户查询的语义，相关性匹配才可能发生。

刚接触搜索的人通常会把分析当成一种从文档中提取一组单词（word）的流程。尽管 token 时常直接相当于文本中的单词，但它的作用可能远不止于此。分析的各个步骤（字符过滤、分词处理、token 过滤）都是在预测用户的意图。如果执行得当，

分析所提取到的文档语义远远胜过一堆单词。带着意图和语义来进行分析，会极大地提升搜索的相关性。

看完这一章你就会明白我们这段话的意思了。例如，也许你会感到惊讶，token 压根就不一定非要和单词相对应！很快你就会看到，有意义的 token（特征）可以从形形色色的各种事物中被提取到，如：地理位置、影像图片，甚至用口哨吹出的旋律。

## 4.2 控制查准率和查全率

查准率（Precision）和查全率（Recall）是搜索相关性的两个基本度量。给定一个查询和由搜索引擎返回的一组文档（结果集），这两个度量的定义如下：

- 查准率——具有相关性的文档在结果集中的百分比。
- 查全率——结果集中被返回的相关性文档的百分比。

必须承认，这两个定义一开始理解起来有点难度，两者听起来好像是一回事。在后续的讨论中，我们会给出一个完整的例子，帮助你理解这两个定义以及它们之间的区别。马上你还会看到，在设计搜索应用时，为什么牢记这两个概念是如此重要。

此外，我们还会演示，查准率和查全率是如何时常彼此掣肘的。一般而言，越是提高查全率，查准率就会变得越低；而越是提高查准率，查全率就会变得越低。这暗示了，对搜索的相关性而言，人们所能达到的最佳效果是受限的。好在我们可以绕开这个限制。让我们在接下来的讨论中一探究竟。

### 4.2.1 查准率和查全率的例子

让我们以一个新的例子开始。这次稍有不同，让我们用——噢，我们所不知道的——水果。在从 4.1.1 节的苹果蛀虫事件回过神来之后，你又回到了水果摊，并且考虑问题更加仔细了。

最初走到水果摊跟前时，你要找的是苹果——更确切地说，你的搜索条件是“红色的、大小适中的水果”。你是通过这些条件得到搜索结果的，如图 4.2 所示。

让我们想一想，如何用查准率和查全率来描述这个结果集。看一下搜索结果，我们得到了 3 个苹果和 3 个红色的、大小适中的，不是苹果的水果（1 个西红柿，1 个柿子椒，1 个石榴）。回顾之前的定义，查准率是正确结果的百分比。在本例中，6 个水果中有 3 个是苹果，因此该结果集的查准率是  $(3 \div 6) \times 100 = 50\%$ 。另外，

仔细查看所有水果之后，我们发现摊位上 13 个可供选择的水果中，有 5 个是苹果。查全率是搜索结果中返回正确结果的百分比。在本例中，水果摊上有 5 个苹果，结果返回了 3 个，搜索苹果的查全率是  $(3 \div 5) \times 100 = 60\%$ 。

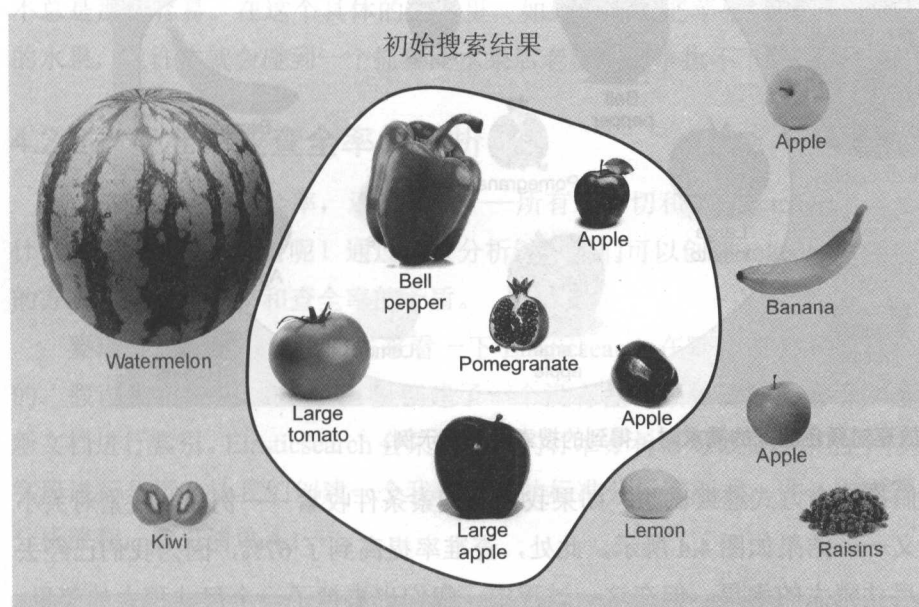


图 4.2 在苹果搜索中所用的文档及其搜索结果的图示

理想情况下，查准率和查全率应该都是 100%。但这几乎是不可能的。不仅如此，查准率和查全率还常常彼此制约。如果改善了查全率，查准率就会受到影响，搜索结果中就会包含错误的内容。另一方面，如果改善了查准率，查全率就会受到影响，搜索结果中就会漏掉一些相关的匹配。

为了更好地理解查准率和查全率彼此矛盾的本质，让我们来看一下这个现象在水果的例子中是怎么表现的。如果我们想改善查全率，就必须把搜索条件放宽一些。为此，假设我们把黄颜色的水果也包含进来会怎样呢？（有些苹果是黄颜色的，不是吗？）如图 4.3 所示，我们的确把另一个苹果也选了出来，这样一来查全率就提高到了 80%。但是因为大多数苹果都不是黄色的，我们也把两个错误的结果选了出来，查准率降到了 44%。



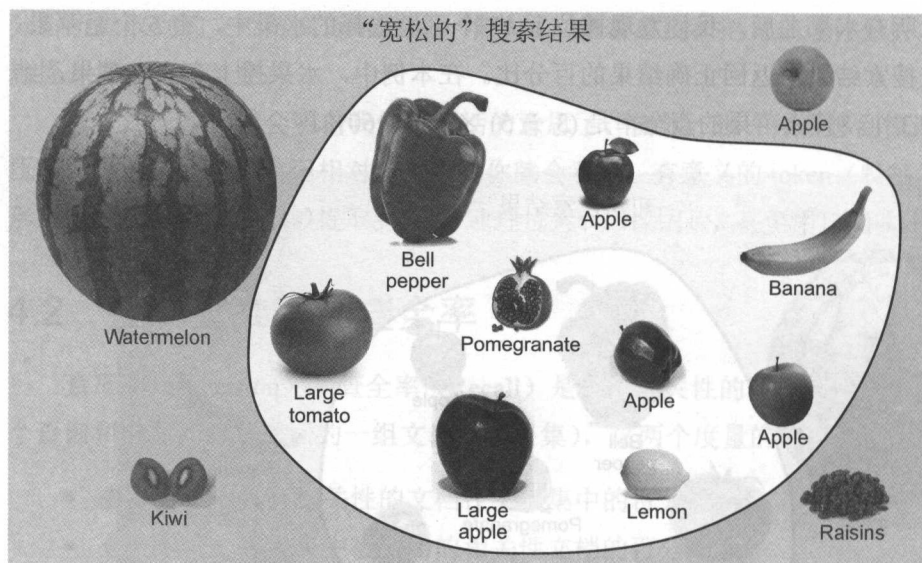


图 4.3 当放宽对颜色匹配的需求时，得到的搜索结果集示例。

让我们换一个方式继续试验。如果我们把搜索条件收紧——例如，收紧对大小适中的定义——结果如图 4.4 所示。此处，查准率提高到了 67%，因为我们已经去掉了两个尺寸稍大的水果。但在这一过程中，我们也去掉了一个尺寸稍大的苹果，查全率降到了 40%。

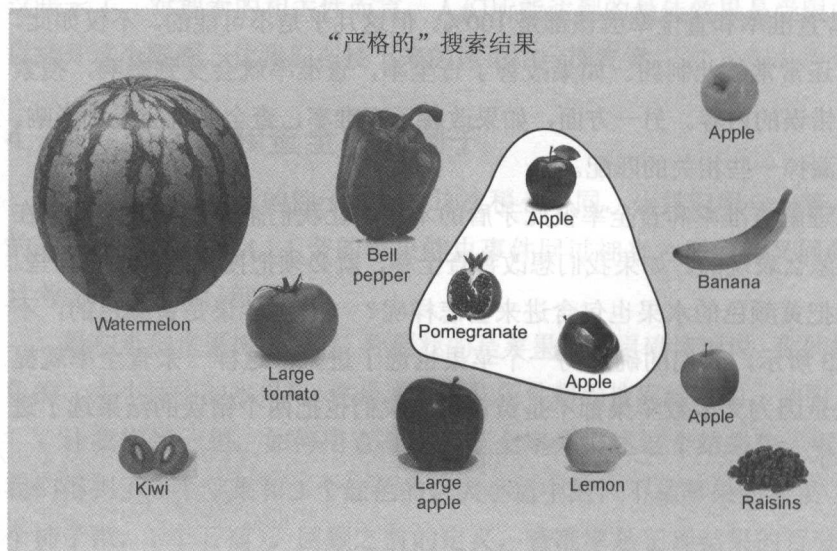


图 4.4 当收紧对尺寸大小的需求时，得到的搜索结果集示例。



尽管查准率和查全率通常彼此制约，我们还是想办法克服这种制约的：寻找更多的特征。如果我们在搜索中再加入一个关于口味的字段，那么西红柿就可以被轻松去掉了，因为它的味道根本不甜。不过，遗憾的是，要找出新的特性放入搜索中，不总是那么容易。在这个具体的例子里，如果你为了把苹果挑出来，决定遍尝所有的水果，也许你就会碰到一个抓狂的水果店老板与你争执不下了！

## 4.2.2 查准率或查全率的分析

可是查准率、查全率，还有水果——所有这一切和文本、token，以及分析又是什么关系呢？关系大着呢！通过调整分析链，我们可以创建 token，以任何我们希望的方式来平衡查准率和查全率的矛盾。

要理解这一点，让我们先来看一下 Elasticsearch 在默认情况下是如何分析文本的。假设我们在 Elasticsearch 里创建了一个没有任何额外设置的索引，然后对一篇新文档进行索引。Elasticsearch 会采用默认的标准分析器对该文档中的字符型(string)字段进行分析。让我们创建一个我们自己的标准分析器副本，进一步来看一下这个分析流程，参见清单 4.1。

### 清单 4.1 重建标准分析器

```
POST my_library
{
  "settings": {
    "analysis": {
      "analyzer": {
        "standard_clone": {
          "tokenizer": "standard",
          "filter": [
            "standard",
            "lowercase",
            "stop"
          ]
        }
      }
    }
  }
}
```

对了，你会注意到在本章我们并没有使用前一章定义的那些函数。我们抛开了 TMDb 数据集。取而代之的，是直接使用 HTTP 命令与 Elasticsearch 进行交互，以便对尽可能多的应用快速进行试验。我们在代码库中的相关示例也反复使用了这些相对底层的命令，不过它们是用 Python 写的。

好了，言归正传！大家可能还记得在第 2 章里，分析分为三步：字符过滤、分词处理和 token 过滤。而且还介绍了，字符过滤有机会对整篇文档进行修改。分词处理将文本切分成一连串 token。最后，token 过滤通过修改、删除或插入 token 对

这一连串 token 加以修改。

在清单 4.1 中，我们的 `standard_clone` 分析器没有用到任何字符过滤器，它用的是标准分词器，然后逐一经过标准过滤器、小写转换过滤器，以及禁用词过滤器，对 token 进行过滤。以下是每一步所做的工作：

- 标准分词器——按空格和标点切分文本；这种方法对大多数欧洲语言来说是没有问题的。相关工具广为流传，成了事实上的标准。
- 标准过滤器——什么也不做！目前被作为一个占位符，应该是当初为了实现与标准过滤器相关联的 token 过滤而放在那里的。
- 小写转换过滤器——将所有字符转成小写。
- 禁用词过滤器——去掉一些出现在禁用词列表里的常用单词。如果我们不喜欢 Elasticsearch 使用的默认列表，也可以自己定义。

现在让我们用 `_analyze` 服务来检验一下由例句生成的 token：

```
GET my_library/_analyze?analyzer=standard_clone&
text="Dr. Strangelove: Or How I Learned to
Stop Worrying and Love the Bomb"
```

在第 3 章中，我们已经展示过了，分析器是可以被用来搜集 token、起止偏移量、token 类型，以及位置信息的，不过这里我们只对 token 感兴趣。在本例中，生成的 token 如下所示：

```
[dr][strangelove][how][i][learned][stop][worrying][love][bomb]
```

和我们期望的一样，token 已经被去掉了标点并转成了小写，而那些常用词也已经被去掉了。

标准分析器通常是构造分析链的一个很好的起点；搜索结果会有较高的查准率，错误结果也很少。但查全率可能会较低，因为用户查询所用的单词一定要和文档中出现的一模一样才行，否则文档是不会被返回的。设想这样一种困境，用户明确自己在寻找一部电影，可就是忘了片名：

```
GET tmdb/movie/_search
{ "query": {
  "multi_match": {
    "query":
      "mr. weirdlove: don't worry, I'm learning to start loving bombs",
    "fields": ["title"]}}}
}
```

查询阶段的分析对其进行了分词处理，结果与我们想的一样：

```
[mr][weirdlove][don't][worry][i'm][learning][start][loving][bombs]
```

可是从查询中得到的 token，没有一个能与根据电影片名生成的 token 相匹配的。搜索引擎存在的意义，就是为了在用户没有掌握全部信息的情况下为其找到内容。否则，它们为什么要存在呢？明明是用户要找的文档，这条查询却无法返回，这一事实说明了，这种特征建模的方法不管用。

我们可以做得更好！再回忆一下，搜索引擎不只是一个复杂的 token 匹配系统。它对 token 的语义一无所知；它只知道如何快速找到包含一组特定 token 的文档。以一种捕获语义的方式对文档进行分词，是相关性技术工程师的职责。原则上，分析过程不应把单词映射为 token；而应把语义和用户意图映射为 token。

让我们以一种与此前不同的分析链来示范上述概念。英语的文本分析器约束较少，有助于确保 token 携带单词的英语语义。和之前一样，Elasticsearch 早已配备了英语分析器，不过为了清晰地展示其工作过程，我们将复制一个英语分析器，参见清单 4.2。这一次我们将深入细节，告诉你如何按需定制分析链，为此我们还复制了 token 过滤器。

#### 清单 4.2 重建英语分析器的细节

```
POST my_library
```

```
{
  "settings": {
    "analysis": {
      "filter": {
        "english_stop": {
          "type": "stop",
          "stopwords": "_english_",
          "english_keywords": {
            "type": "keyword_marker",
            "keywords_path": "/tmp/keywords.txt",
            "english_stemmer": {
              "type": "stemmer",
              "language": "english",
              "english_possessive_stemmer": {
                "type": "stemmer",
                "language": "possessive_english"
              }
            }
          }
        },
        "analyzer": {
          "english_clone": {
            "tokenizer": "standard",
            "filter": [
```

```
"english_possessive_stemmer",
"lowercase",
"english_stop",
"english_keywords",
"english_stemmer"]]]]]}
```

让我们花一点时间来了解一下，英语分析器的每一步分析都做了些什么：

- 标准分词器——此处我们依然选择标准分词器，因为它对欧洲语言的分词效果很好。
- english\_possessive\_stemmer 过滤器——去掉单词末尾的 s。
- 小写转换过滤器——将所有字符转换成小写。
- english\_stop 过滤器——去掉那些常用的英语单词。
- english\_keywords 过滤器——“保护”单词不被后续的词干处理环节破坏掉（稍后会有详细的讨论）。
- english\_stemmer 过滤器——对单词的词尾进行标准化处理，从而像 walking 和 walked 这样的单词会被映射为同一个 token。

正如我们看到的那样，标准分析器和英语分析器所做的工作在很多地方都是一样的。英语分析器的“特制调味料”<sup>1</sup>在 token 过滤器 english\_stemmer 里。该过滤器从前面的分析步骤中得到单词 token，并对其进行了标准化处理，把具有相同词干的单词映射为同一个 token。例如，如下所示，所有这些词都映射成了名为 flower 的 token：

```
GET my_library/_analyze?analyzer=english_clone&
text="flower flowers flowering flowered flower"
```

类似的，如下所示，所有这些词都映射成了名为 silli 的 token：

```
GET my_library/_analyze?analyzer=english_clone&
text="silly silliness sillied sillying"
```

你也许注意到了，在前面的例子里并非所有单词都是真正的单词。但是 english\_stemmer 并不介意。它没有借助字典来判断单词的词根；而是使用了一种启发式的方法将单词映射为相应的词根形式。启发式方法准确度高，因此很有用，

<sup>1</sup> 即与标准分析器的不同之处。——译者注

但有时也会出错。例如，单词 **Maine**（美国的州名）会被词干化成 **main**，从而被误当作普通的英语单词 **main**。如果这对你而言是一个问题，别慌！可以把有问题的单词加入关键字文件（清单 4.2 中的 `/tmp/keywords.txt`），告诉所有后续的词干处理逻辑，不要对其进行处理。

回到我们讨论的主题，词干处理是一项为提高查全率而牺牲查准率的特征建模技术。利用上述英语分析器，我们那位健忘的老兄在查找电影时的运气就会更好一些。让我们来演示一下，看一看由准确的片名和不准确的查询生成得到的 **token**，如清单 4.3 所示。

#### 清单 4.3 利用英语分析器对查询和文档的文本进行分词

```
GET my_library/_analyze?analyzer=english_clone&
text="Dr. Strangelove: Or How I Learned to Stop Worrying
and Love the Bomb"

[dr][strangelov][how][i][learn][stop][worri][love][bomb]

GET my_library/_analyze?analyzer=english_clone&
text="mr. weirdlove: don't worry, I'm learning to start loving bombs"

[mr][weirdlov][don't][worri][i'm][learn][start][love][bomb]
```

如粗体处的文本所示，利用英语分析器，我们得到了 4 个匹配的 **token**，而用标准分析器则找不到任何 **token** 与之匹配。原来无法返回的那些文档，现在在搜索结果里有了较高的排名。

### 4.2.3 一味提高查全率

通过示例我们已经清楚地看到，牺牲查准率以提高查全率是一种行之有效的提高整体搜索相关性的方法。但是要当心，这样做也可能会误入歧途。让我们再用一个分析器——语音分析器（**phonetic analyzer**），来演示这一点。它会根据发音将单词映射到相应的 **token**。我们将利用语音分析器来构建一个能有效应对拼写错误的搜索程序。表面上看起来这是一件好事，因为它提高了查全率，让文档更容易被搜到，即使文档或查询中包含拼写错误也没有关系。不过接下来我们会为大家演示，这种做法经常会过于粗暴。

对于这个演示，我们需要从头开始构建分析器，因为 Elasticsearch 自己并不提

供预置的语音分析器。我们首先来安装语音处理插件<sup>1</sup>，进入终端，转到 `elasticsearch` 目录，在命令行提示符下输入：

```
bin/plugin install analysis-phonetic
```

一旦插件安装完毕，我们就可以为 `Elasticsearch` 提供如下所示的配置，创建一个语音分析器，如清单 4.4 所示。

#### 清单 4.4 构建语音分析器

```
POST my_library
{ "settings": {
  "analysis": {
    "analyzer": {
      "phonetic": {
        "tokenizer": "standard",
        "filter": [
          "standard",
          "lowercase",
          "my_doublemetaphone"
        ]
      },
      "filter": {
        "my_doublemetaphone": {
          "type": "phonetic",
          "encoder": "doublemetaphone",
          "replace": true
        }
      }
    }
  }
}
```

此处我们应该能够辨认出几个熟悉的组件：标准分词器、标准过滤器，还有最后的小写转换过滤器。新的 `my_doublemetaphone` 过滤器是这次的焦点，所有引人注目的事情都是在这里发生的。该过滤器接收经小写转换后的单词 `apple` 和 `banana`，将它们相应转换成代表基本发音的 `token`，在本例中即 `APLS` 和 `PNNS`。如我们所见，分析器野心很大——所有除元音以外的字母都被去掉了，一些像 `ba` 和 `pa` 这样的发音被映射成了同一符号（本例中为 `P`）。不过语音分析过程还是如其所宣称的那样工作得很好，因为一个拼写存在严重错误的搜索“`oopuls`”和“`banunus`”的确会和包含正确拼写的单词 `apple` 和 `banana` 的文档相匹配。

但是，有一个问题。我们在实践中有一个有关分析流程的重要经验教训：

尽可能在一定程度上，确保 `token` 不仅代表单词，还要代表语义。

<sup>1</sup> 有关语音处理插件的更多信息可参见<https://github.com/elastic/elasticsearch/tree/master/plugins/analysis-phonetic>。



这就是为什么我们时常将 token 的词干处理纳入分析当中，将其作为一种良好实践的原因。词干处理把单词的多种表现形式统一成了一种唯一的形式——例如在之前的例子里，将 flower、flowers、flowering 还有 flowered 都转换成了统一的词根形式 flower。如果没有词干处理，这些单词就会代表完全不同的事物。搜索引擎将无法识别出它们所代表的是同一个意思：搜索 flowering 不会与 flowers 匹配。因此，如果没有词干处理，那么查全率就会大幅降低。查准率也许会提高，但对于一个典型的搜索用户而言，这一点点提高是没有多大用处的。

另一方面，语音分词是冒着降低查准率的风险，最大化查全率。此处我们再想一想分析是如何将 token 和语义对应起来的。在语音分词器的作用下，与每个 token 相关联的语义都太宽泛了。许多不同的语义都被映射成相同的 token，造成查准率如此之低。

## 4.3 查准率和查全率——让鱼和熊掌兼得

前途看似暗淡。作为一名相关性技术的研发工程师，我们面临着一个艰难的抉择：为了让文档能够被索引，必须对文本进行分析。但是我们所选择的分析方法，将相关性的表现“钉”在了查准率 - 查全率图谱（precision-recall spectrum）上的某个固定的位置。

幸好，搜索技术的另一面，让相关性技术工程师能够打造出查全率和查准率都不错的搜索系统。在第 2 章中，我们了解了搜索引擎的原理。描述搜索引擎的另一种说法如下所示：

搜索引擎不过是一个复杂的 token 匹配系统加文档排名系统。

就是这后半部分，规避了查准率 / 查全率这一对根本性矛盾。相关性排名就是对系统施以引导的方法。它帮助我们提高前  $N$  个——可能对用户最为重要的——结果的查准率。你将会看到，我们所采用的技巧就是，对分析加以微调，严格控制何时应将 token 的各种不同形态映射为同一语义。

### 4.3.1 评价单一字段中特征的强度

相关性排名究竟是怎样规避查准率 / 查全率这一对矛盾的呢？分析不仅控制着

匹配，还控制着  $TF \times IDF$  相关性排名，从而更为精确地反映出某一给定特征在文本（或可以被分词的任何形式的数据）中的强度（strength）。

还记得我们的好朋友  $TD \times IDF$  吗？词频（TF）计算的是搜索词在文档中出现的次数。文档频率计算的是在整个文档集中包含特定词的文档个数（IDF 是其倒数）。这两个因素共同作用，对匹配到的文本就搜索相关性进行评价。

不过，TF 和 IDF 能对同一语义的不同形态进行统计吗？举个例子，如果你把一位说英语的朋友拽到一段文字面前，让他统计出 run 在文中出现的次数，他会怎么做呢？也许他会把各种形态的 run（running、runs、run，也许还有 ran）都加到一起。即使所有形态的 run 都在文中被用到了，这位朋友可能还是会认为这段文字和 running 的关系最为密切。他可能会说，即使 run 的所有形态都被用到了，running 这个特征在文中被多次提到，在对“run”的搜索中理应给它打更高的分。

正如我们已经看到的那样，机械呆板的搜索引擎在没有人帮忙的情况下是不会把这些不同形态的 run 当作同一个单词来统计的。当文本中存在单词的替代形态时，每一种形态都会被单独统计和评价。搜索引擎在做  $TF \times IDF$  评价时是不会知道这里有一个“run-ness”的强特征（strong feature）的。它所看到的是孤立的，在文中被略微提及的各种不同的单词：run、running，还有 runs。所幸的是，我们已经看到了，分析是如何帮助我们将所有这些单词形态统一标准化成一个单词的，这样一来就可以对其进行统计和评价了。

我们来演示一下相关性排名和分析流程是如何协同工作，对查准率和查全率进行微调的。让我们从一个简单的例子开始——一条查询，只涉及一个字段里的一个单词，使用默认的标准分析器对我们的测试索引，my\_library，进行处理：

```
POST my_library
{"settings": {
  "number_of_shards": 1}}
```

然后我们就可以对文档进行索引了：

```
PUT my_library/example/1
{ "title": "apple apple apple apple apple" }
PUT my_library/example/2
{ "title": "apple apple apple banana banana" }
PUT my_library/example/3
{ "title": "apple banana blueberry coconut" }
```

最后发起一个简单的搜索请求，在 title 字段里填入 apple：

```
GET my_library/example/_search
{ "explain": "true",
  "query": {
    "match": {
      "title": "apple"}}
```

这里给出了针对这一查询的前三个返回的文档，并附有相应的解释 (explain)

文本：

1. {u'title': u'**apple apple apple apple**'}
  - 0.3001879, weight(title:apple in 0) [PerFieldSimilarity], result of:
    - 0.3001879, fieldWeight in 0, product of:
      - 2.236068, tf(freq=5.0), with freq of:
        - 5.0, termFreq=5.0
      - 0.30685282, idf(docFreq=1, maxDocs=1)
      - 0.4375, fieldNorm(doc=0)
2. {u'title': u'**apple apple apple** banana banana'}
  - 0.23252454, weight(title:apple in 0) [PerFieldSimilarity], result of:
    - 0.23252454, fieldWeight in 0, product of:
      - 1.7320508, tf(freq=3.0), with freq of:
        - 3.0, termFreq=3.0
      - 0.30685282, idf(docFreq=1, maxDocs=1)
      - 0.4375, fieldNorm(doc=0)
3. {u'title': u'**apple** banana blueberry coconut'}
  - 0.15342641, weight(title:apple in 0) [PerFieldSimilarity], result of:
    - 0.15342641, fieldWeight in 0, product of:
      - 1.0, tf(freq=1.0), with freq of:
        - 1.0, termFreq=1.0
      - 0.30685282, idf(docFreq=1, maxDocs=1)
      - 0.5, fieldNorm(doc=0)

搜索引擎的启发式  $TF \times IDF$  评价方法，根据文档是否显著反映搜索词的特征，对其进行排序。之前我们曾经讨论过购买水果的例子：根据水果是否有光泽或是红色，判断其是否是红蛇果。例子所考虑的这些特征都是二元的。水果是红色的吗——是或否？它有光泽吗——是或否？实际上，像“redness”这样的特征，是分布在一个从“丝毫不红”到“略带粉红”再到“如消防车一般鲜红”这样的图谱上的。

根据  $TF \times IDF$  的值，那些涉及 apple 的文档也分布在这个图谱上。一些匹配的文档和 apple 紧密相关。这些文档里 apple 的词频较高，我们称之为 apple-y 型文档，

它们很少提到别的概念。而另一些文档则提到了别的单词，“apple”在里面只是一个次要概念，这些文档完全不是 apple-y 型的。正因为如此，其  $TF \times IDF$  评价体现出来的 apple-y 特征较弱。利用相关性排名，至少对前  $N$  个结果而言，我们的查准率得到了提升。

更为重要的是，分析过程可以通过对某个想法的不同表现形式采取标准化处理，从而改进  $TF$  和  $IDF$  的计算！这样做可以提高评价的准确性，使其与用户对相关性的直觉认识保持一致。

例如，我们目前的搜索明显有一个相关性排名的 bug！假设我们给一篇新的文档建索引：

```
PUT my_library/example/4
{ "title": "apples apple" }
```

如果该文档在搜索 apple 时得到了相对较低的  $TF \times IDF$  评价，你也许会感到惊讶。重复之前的搜索，我们会发现，尽管该文档 100% 由涉及 apple 的单词构成，可它还是位于所有文档搜索结果中靠近末尾的地方：

```
3. { 'title': 'apples apple' }
0.48553526, weight(title:apple in 0) [PerFieldSimilarity], result of:
0.48553526, fieldWeight in 0, product of:
  1.0, tf(freq=1.0), with freq of:
    1.0, termFreq=1.0
    0.7768564, idf(docFreq=4, maxDocs=4)
    0.625, fieldNorm(doc=0)
```

注意粗体部分，apple 的频率是 1。可是文中提到 apple-y 形式的东西有两次啊！为什么  $TF \times IDF$  对其视而不见呢？

你已经知道是怎么回事了吧。因为我们用的是标准分析器，对 apple 的查询是不会与 apples 相匹配的。不管你那位说英语的朋友是怎么想的，搜索引擎并不理解这些说法其实是一个意思。利用分析技术对同一概念的多个等价形式进行管理，按需对其进行标准化（normalizing）和差异化（discriminating）处理，从而让搜索引擎能在评价时做出正确的区分（differentiations），这是你作为一名相关性技术工程师的职责所在。

简单地利用英语分析器，通过将各种形式的 apple 标准化成词干 appl，就可以改进  $TF \times IDF$  评价。在使用了英语分析器并重建索引之后，上述文档在搜索结果中上升到了第二位，对于词干 appl 的词频也增加到了 2：

```

2. {u'title': u'apples apple'}
0.6866506, weight(title:appl in 3) [PerFieldSimilarity], result of:
0.6866506, fieldWeight in 3, product of:
1.4142135, tf(freq=2.0), with freq of:
2.0, termFreq=2.0
0.7768564, idf(docFreq=4, maxDocs=4)
0.625, fieldNorm(doc=3)

```

`token` 代表了特征在文本中的重要程度。前面我们已经见识过在做决策时查准率与查全率之间看似不可避免的权衡。但是如我们所见，控制相关性排名可以帮助降低这种权衡的必要性。通过分析，我们有能力在提高查全率的同时提高  $TF \times IDF$  的相关性分值！分析技术赋予了我们这样的力量，可以按照期望的区分度进行匹配并进行  $TF \times IDF$  评价。

### 4.3.2 超越 $TF \times IDF$ 的评价：多搜索词与多字段

和上一个例子不同，用户搜索时很少只用一个搜索词。他们会在搜索中使用多个单词。我们应该利用这一点来改进查准率！幸好，协调因子（`coord`）是我们的“好帮手”。前面我们在讨论 Lucene 的布尔搜索（Boolean search）时曾经谈到过 `coord`，不过此处我们想指出的是其在偏置评价（biasing scoring，即搜索词被提及次数越多，打分越高）中所扮演的角色。

如果我们重复之前对 apple banana 的搜索，`coord` 会以最终分值乘以  $1/2$  的方式，对单纯匹配 apple 的情况加以评判。因为两个搜索词里只有一个与文档相匹配，所以在总体上对其进行了评判。我们可以在针对 apple banana 查询的这些解释信息的片段中观察到这一点（为简化起见略去了  $TF \times IDF$ ）：

```

1. {u'title': u'apple apple apple banana banana'}
0.9862758, sum of:
0.30409503, weight(title:appl in 1) [PerFieldSimilarity], result of:
0.68218076, weight(title:banana in 1) [PerFieldSimilarity], result of:

3. {'title': 'apple apple apple apple apple'}
0.1962925, product of:
0.392585, sum of:
0.392585, weight(title:appl in 0) [PerFieldSimilarity], result of:
0.5, coord(1/2)

```

在后续几章中探讨查询策略时，我们会更加清楚地看到，在包含多项匹配的搜索结果中突出其中部分匹配项时，`coord` 所扮演的重要角色。

规避查准率 / 查全率权衡的另一个办法是借助多字段搜索，每个字段采用不同形式的分析手段。不同分析之间看起来也许是彼此互斥的：标准分析、词干分析、语音分析。但是如果愿意的话，我们可以把它们组合在一起使用。用不同分析器对同一段文本进行处理，将结果放到不同字段里，这样做是很常见的。在第5章及其以后的章节中，我们会看到多个字段是如何贡献出它们自己的信号，改进整体排名函数和改进查准率 / 查全率这一对矛盾的。

## 4.4 分析策略

本章至此，我们已经学习了一系列利用分析手段来控制相关性的重要原则。首先，我们明白了查准率和查全率的基本原则。我们还讨论了这两个概念之间的相互作用：提升查准率或查全率中的任何一项，通常都会导致另一项的下降。但是在针对文档评价的讨论中，我们也看到了，同时保持良好的查全率和查准率是能做到的，因为搜索引擎在评价时往往会把相关性最高的文档排在搜索结果的最前面。

我们还为分析决策的执行制订了若干指导性原则。首先，token 不应该只被映射成文档中的词汇，而是要尽可能映射成词汇的语义。这一点我们已经在 4.2.2 节里看到了。在那一节中，我们解释了词干处理如何帮助确保 token 能适度捕获单词的语义细节。其次，分析技术为相关性技术工程师提供了机会，可以预测搜索用户的行为与意图。在 4.2.3 节我们描述了语音分词是如何预测用户偶尔把搜索词拼错这种情况的。

在本节中，我们更加有针对性地给出了一系列利用分析来修改和控制相关性的例子。这样的例子不胜枚举；要想囊括读者可能遇到的各种搜索问题，那是不可能的。相反，我们在后面几节里会给出几个小例子。当遇到极具挑战的问题时，通过恰当的分析，对系统能得到什么样的结果，会有我们自己更好的想法。在这些例子中，有一部分是专门为了演示某些相关性原则的。另一部分则聚焦于分析中用到的某些特殊的特征，这些特征可以被广泛用于各类相关性问题。

### 4.4.1 处理分隔符

我们之前的例子所使用的字段都是由文本构成的，包含以空格或标点分割的一系列单词。不过情况并非总是如此，而且分隔符如果处理不当可能会导致搜索结果不尽如人意。考虑下面这几个例子，它们关注的是缩略词和电话号码。



## 缩略词

有时缩略词用的是句号，有时却未必——例如，I.B.M. 和 IBM。分析过程必须对缩略词进行标准化处理，这样不管缩略词是何种形式，其生成的 token 总是一样的。用 word\_delimiter 过滤器可以轻松搞定这件事情。下面这个分析链就可以生成合适的 token：

POST example

```
{
  "settings": {
    "analysis": {
      "filter": {
        "acronyms": {
          "type": "word_delimiter",
          "catenate_all": true,
          "generate_word_parts": false,
          "generate_number_parts": false}},
        "analyzer": {
          "standard_with_acronyms": {
            "tokenizer": "standard",
            "filter": ["standard", "lowercase", "acronyms"]}}}}}
```

上述分析链建立了一个类型为 word\_delimiter 的缩略语过滤器。该 token 过滤器是被设计用来在各种边界处对 token 进行拆分的，举例来说，这种边界包括像标点符号的变化 (Wi-Fi)、大小写的变化 (wiFi)，还有数字与非数字之间的变化 (2016AD)。我们并没有把单词中被分割的部分各自变成 token，而是选择了更好的方式。此处，我们没有选择 generate\_word\_parts（否则，I.B.M. 就会变成三个 token，i、b、m）。我们还选择了 catenate\_all，意思是不管被分割后生成的是什麼，我们都会把它们合并起来作为一个 token。测试一下可以发现，这样做到了我们预期的效果：

GET example/\_analyze?analyzer=standard\_with\_acronyms  
&text=I.B.M. versus IBM versus ibm

```
[ibm] [versus] [ibm] [versus] [ibm]
```

因此现在，用户在搜索“IBM”或“I.B.M.”时会找到同样的匹配。但是，要注意，当缩略词碰巧是另一个单词的时候，这样做是会有问题的（比如 N.E.W.，代表 National Engineering Week）。在这个例子里，产生的 token 就会是 new，这是一个非常糟糕的文本特征，因为它同时含有两层意思：National Engineering Week 和

些“不太旧的”东西。为了帮助解决这个问题，我们给缩略词过滤器加上了“`preserve_original`”: `true`。这样一来，`N.E.W.` 就会被同时转换成 `new` 和 `n.e.w.` 两个 token。在这种情况下，搜“`new`”会偶尔匹配含缩略词 `N.E.W.` 的文档。这样是可以的，因为也许用户想要的就是这个，而非去掉点号的结果。还是这个例子，搜索“`N.E.W.`”会精确匹配那个被保留下来的原始 token，相关性得到了极大改善。

## 电话号码

电话号码——例如，`1(800)867-5209` 和 `1.800.867.5309`——则更为复杂。虽然我们很少希望把电话号码中的分隔符去掉（人们通常不愿意看到号码被写成 `18008675309` 这样的形式），但电话号码中用到的分隔符种类繁多，用户在搜索电话号码时可能会选择其中任何一种格式。

一如之前的例子所示，此处，`word_delimiter` 通过有效地删除分隔符，能帮我们再一次渡过难关。不过，这里有一个小小的调整。前面那两个示例号码都会生成像 `18008675309` 这样的 token，但是我们必须预测用户期望的行为。用户总是输入完整的电话号码，包括区号和国家代号吗？也许不是。因此，让我们再加一个过滤器，将这种情况也考虑进去，生成两个额外的 token：一个是后 7 位数字（本地号码），一个是后 10 位数字（长途号码）：

```
POST my_library
{
  "settings": {
    "analysis": {
      "filter": {
        "phone_num_filter": {
          "type": "word_delimiter",
          "catenate_all": true,
          "generate_number_parts": false},
        "phone_num_parts": {
          "type": "pattern_capture",
          "patterns": ["(\\d{7}$)", "(\\d{10}$)"],
          "preserve_original": true}},
      "analyzer": {
        "phone_num": {
          "tokenizer": "keyword",
          "filter": ["phone_num_filter", "phone_num_parts"]}}}}}
```

与前例相比，这里最为重要的不同，并非分析链定义中被突出显示的那几处。首先要注意的一点是，现在的分词器已经不是标准分词器了，而是关键词分词器。

此处假设，我们处理的是 `phone_number` 字段，而非 `free-text` 字段——即恰好包含了电话号码的自由文本。关键词分词器不会把输入内容切分成多个 `token`，而是生成一个单一的 `token`，把字段中未经修改的整段文本都包含进来。“可为什么要这样呢？”你会问。“难道我们就不能选择压根儿不分析这个字段吗？”好问题。我们之所以使用关键词分词器，是因为它允许我们紧接着用后续过滤器对 `token` 做进一步修改。

在本例中，`phone_num_parts` 是紧随其后的一个过滤器。这是一个包含两种模式 (`pattern`) 的 `pattern_capture` 过滤器，一个用于捕获输入的后 7 位数字 (本地号码)，一个用于捕获后 10 位数字 (长途号码)。原始 `token` 也同时被保留了下来。让我们做一下测试：

```
GET example/_analyze?analyzer=phone_num&text=1(800)867-5309
```

```
[18008675309][8008675309][8675309]
```

基于上述分析方案，用户对“800.867.5309”的搜索会被分词成 8008675309 和 8675309，而这两个 `token` 都会与包含号码 1(800)867-5309 的文档相匹配。完美！

在前面两个例子中，我们不断调整分析，目的就是为捕获字段中所含数据的最为重要的特征——语义。在这种情况下，缩略词和电话号码的语义都不在它们的分隔符中，而在构成它们的字符和数字里。除此以外，我们预测的用户行为，可能会使用任何形式的分隔符，或者甚至根本不用分隔符。对电话号码而言，捕获号码中一个有意义的子集，是对用户意图的一种建模方法，它有效地承认了用户既可能会搜索本地号码，也可能会搜索完整号码的事实。我们希望尽可能给出最为相关的结果，不论用户使用的是什么格式的号码。

#### 4.4.2 捕获同义词的语义

同义词在解决一些边界情况 (`corner-case`) 下的相关性问题时常常能派上用场。考虑一个在线服装零售商遇到的麻烦，我们正在试图解决查询“dress shoes”时遇到的问题。对我们而言，“dress shoes”也许会让大家联想到男士穿的皮鞋。但是某位刚入行的搜索工程师也许会惊讶地发现，搜索“dress shoes”会得到满满一屏背心裙 (`sundresses`) 和网球鞋 (`tennis shoes`) 的页面！解决办法仍然是确保 `token` 捕获语义而不只是词汇。

我们如何捕获同义词的语义呢？在英语中，任何时候只要单词 `dress` 后面紧跟 `shoes`，我们就是在谈论一个专门的概念：`dress shoes`。注意到这一点，我们就可以

借助分析将这一概念映射到一个单独的 token 上，就像下面这样：

```
POST retail
{
  "settings": {
    "analysis": {
      "filter": {
        <english filters omitted>
        "retail_syn_filter": {
          "type": "synonym",
          "synonyms": [
            "dress shoe,dress shoes => dress_shoe, shoe"
          ]
        },
        "analyzer": {
          "retail_analyzer": {
            "tokenizer": "standard",
            "filter": [
              "english_possessive_stemmer",
              "lowercase",
              "retail_syn_filter",
              "english_keywords",
              "english_stemmer"
            ]
          }
        }
      }
    }
  }
  "mappings": {
    "items": {
      "properties": {
        "desc": {
          "type": "string",
          "analyzer": "retail_analyzer"
        }
      }
    }
  }
}
```

来自清单4.2的英语过滤器

① 我们的新分析器

② 此处即过滤器的用武之地

③ 我们的desc字段包含了同义词

在这段被标注过的程序清单里，我们做了下列几件事：

- 创建一个 retail\_syn\_filter ① 过滤器，为 dress shoes 定义一行同义词。
- 在新的 retail\_analyzer ② 分析器里使用上述过滤器。
- 确保上述分析器被用于 desc 字段中的文本 ③。

（注意，我们假设 english\_\* 过滤器的定义与清单 4.2 的相似。）

有一件事情需要考虑，那就是 retail\_syn\_filter 安放的位置。把它放在所有格词干处理、复数词干处理，以及小写转换处理的后面，就能让到达同义词过滤器的输入，得到一定程度的标准化处理，从而生成我们期待中的同义词。试想一下，如果 retail\_syn\_filter 是第一个过滤器，那么 dress shoe's 和 Dress shoes 是不会被同义词过滤器所接纳的。不过，把同义词过滤器放到更为激进的标准化处理前面，这一点也很重要，比如词干或语音的映射，因为这些过滤器改变 token 的程度是如此之大，以至于我们会找不到匹配的同义词。

接下来，让我们来看一下定义同义词的那一行：

```
"dress shoe,dress shoes => dress_shoe, shoe"
```

这一行定义的意思是，任何时候当你看到 `dress shoe` 或 `dress shoes` 时，都要将其映射成两个 `token`：`dress_shoe` 和 `shoe`。此处将 `shoe` 这一点包含进来很重要，因为 `dress shoe` 的确是一双鞋，而且我们希望在搜“`shoes`”时能将这两个 `token` 都包含进来。

让我们找几个文档来建一下索引，看看它是如何工作的：

```
POST retail/items/1 # "dress shoe" document
{ "desc": "bob's brand dress shoes are the bomb diggity"}
POST retail/items/2 # "dress" document
{ "desc": "this little black dress is sure to impress"}
POST retail/items/3 # "shoe" document
{ "desc": "tennis shoes... you know, for tennis"}
```

然后执行如下查询：

```
GET retail/items/_search?
{ "query": {
  "match": {
    "desc": "dress"}}
```

上述查询返回了一条结果，即带 `dress` 的文档。非常好。接下来我们再执行一条类似的查询，查一下“`shoes`”。和期望的一样，这一次我们同时收到了带 `dress shoe` 的文档和带 `shoe` 的文档。这也是我们期望的结果。最后一项测试，做完就收工：查一下“`dress shoes`”，预计只收到带 `dress shoe` 的文档……结果……嘿！为什么带 `dress shoe` 的文档和带 `shoe` 的文档都出现在返回结果里呢？

借助 `_analyze` 服务，找到问题的原因不是太难。下面是这两个疑义文档的分析情况：

- 带 `dress shoe` 的文档：`bob`, `brand`, `dress_sho`, `shoe`, `bomb`, `diggiti`
- 带 `shoe` 的文档：`tenni`, `shoe`, `you`, `know`, `tenni`

别忘了，查询也会接受分析，并生成用于匹配索引的 `token`。在本例中，我们得到了如下 `token`：

```
dress shoe search: dress_sho, shoe
```

正如我们所见到的，有一个事实，它的含义很容易被忽视，那就是分析在索引阶段和搜索阶段都是会发生的。我们对 dress shoes 的同义词扩展处理导致其生成了两个 token——其中一个是 shoe，在那两个疑义文档中也出现了。现在问题的原因显而易见了。

幸好，有一个简单的解决方法。Elasticsearch 允许索引阶段和查询阶段使用不同的分析器。清单 4.5 所示的是利用这一点来解决上述问题的方法。

清单 4.5 为解决 dress shoes 问题而对索引所做的配置

```
POST retail
{ "settings": {
  "analysis": {
    "filter": {
      "retail_syn_filter_index": { # 1. new filter
        "type": "synonym",
        "synonyms": ["dress shoe,dress shoes => dress_shoe, shoe"]},
      "retail_syn_filter_search": {
        "type": "synonym",
        "synonyms": ["dress shoe,dress shoes => dress_shoe"]}},
    "analyzer": {
      "retail_analyzer_index": { # 2. new analysers
        "tokenizer": "standard",
        "filter": [
          "english_possessive_stemmer",
          "lowercase",
          "retail_syn_filter_index",
          "english_stop",
          "english_keywords",
          "english_stemmer"]},
      "retail_analyzer_search": {
        "tokenizer": "standard",
        "filter": [
          "english_possessive_stemmer",
          "lowercase",
          "retail_syn_filter_search",
          "english_stop",
          "english_keywords",
          "english_stemmer"]}}}},
  "mappings": {
    "items": {
      "properties": {
        "desc": {
          "type": "string",
          # 3. two-sided analysis
          "analyzer": "retail_analyzer_index",
          "search_analyzer": "retail_analyzer_search"}}}}}
```

1 索引阶段的同义词过滤器（与清单4.4相同）

2 查询阶段的同义词过滤器，搜索 dress shoe 时强制采用基于 dress\_shoe 的精确搜索

为搜索和索引建立不同的分析器

为搜索和索引建立分析器

为字段设置使用不同的索引和搜索分析器

为字段设置使用不同的索引和搜索分析器

也许看起来有点复杂，不过我们所做的全部工作只是把分析拆成了索引阶



段①（在该阶段，`etail_syn_filter_index` 为 `dress shoes` 生成了两个 token：`dress_shoe` 和 `shoe`）和查询阶段②（在该阶段，`dress shoes` 只对应了一个 token，`dress_shoe`）。如果再执行一遍之前的查询，会发现结果就是我们预计的那样。查询“`dress`”返回的是带 `dress` 的文档，查询“`shoes`”同时返回带 `dress shoe` 和 `shoe` 的文档，查询“`dress shoes`”只返回带 `dress shoe` 的文档。现在搜索工作正常了，因为我们已经准确地捕获了“`dress shoe`”的语义，并以一种能够返回相关性文档的方式，将其编入了搜索引擎。

### 4.4.3 在搜索中为专指性建模

如果我们搜索一样东西，比如“`dog`”，能够返回包含单词如 `poodle`、`terrier` 和 `beagle` 这样的文档，甚至没有用到单词 `dog` 的文档，这样是不是很好？再进一步。搜索“`animal`”，返回的文档中带有 `dog`、`poodle`、`cat` 之类的单词，甚至单词 `animal` 都没有包含在文档里，那样是不是很好？嗯，可以的！通过使用非对称分析（`asymmetric analysis`）技术，我们可以将一种所谓专指性（`specificity`）的概念编入搜索应用中。非对称分析意味着，用于查询阶段的分析和用于索引阶段的分析是不一样的。

### 4.4.4 利用同义词为专指性建模

怎样才能做到这一点呢？其实，在前一小节的例子里我们已经悄悄地展示过了。我们不但利用同义词保证了 `dress shoes` 被分词成一个完整的语义单元（`dress_shoe`），而且还生成了一个额外的 token，即 `shoe`，这样一来，`dress shoes` 依然会匹配对 `shoe` 的搜索。这种分析就是非对称的。在索引分析期间我们生成了两个 token。在查询期间只有一个 token 被生成，`dress_shoe`，这样我们的搜索就可以严格匹配 `dress shoes` 了。

为了透彻理解非对称分析是如何被用来对特指性进行编码的，让我们来看一个通俗一点的例子。此处我们也可以使用前面那个 `dog` 的例子，不过那个例子有点乏味。让我们选择一个更有意思的例子吧，比如，我们所不知道的，水果！考虑一下水果的继承结构，如图 4.5 所示。

基于这一继承结构，我们可以在同义词文件中添加如下记录：

```
apple => apple, fruit
```

```
fuji => fuji, apple, fruit
mcintosh => mcintosh, apple, fruit
gala => gala, apple, fruit
banana => banana, fruit
orange => orange, fruit
```

请注意，同义词文件是将词汇映射为语义上等价或更为宽泛的词。我们将其称为语义扩展（semantic expansion）。例如，fruit 及 apple 在语义上和 apple 是等价或更为宽泛的。

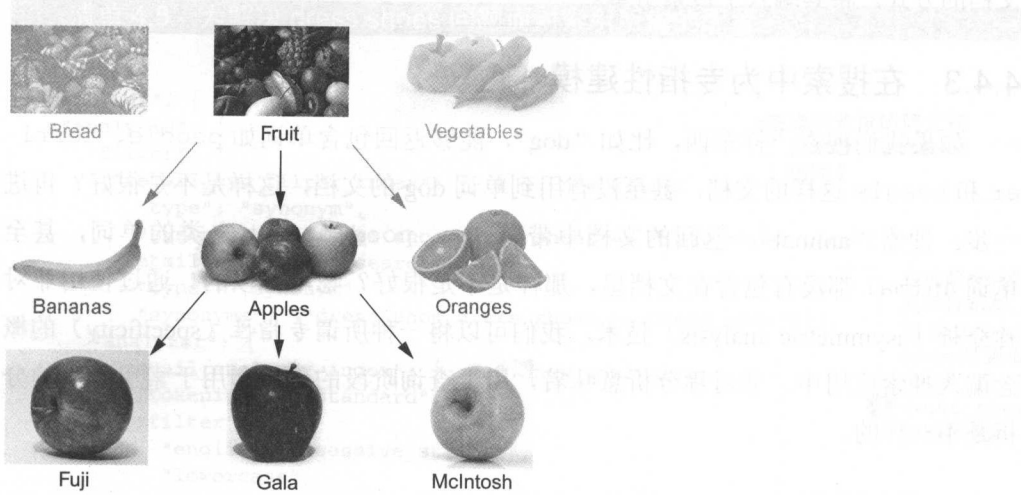


图 4.5 以苹果为核心的水果层级结构

接下来，设想一下，如果我们只在索引阶段使用同义词分析，如表 4.1 所示，搜索行为会是什么样的。如你所见，词汇在索引阶段的同义词匹配对应的是语义上等价或更为宽泛的值，这是我们期望的行为；而查询所匹配的，则是语义上等价或更为具体的文档。例如，对“apple”的查询，匹配的不只是带 apple 的文档，还有带 fuji 的文档。

表 4.1 索引阶段的语义扩展

索引		查询		
文档包含了	生成的 token	Fruit	apple	fuji
Fruit	Fruit	匹配	未匹配	未匹配
apple	apple, fruit	匹配	匹配	未匹配
Fuji	fuji, apple, fruit	匹配	匹配	匹配
orange	orange, fruit	匹配	未匹配	未匹配

反过来设想一下,如果我们只在查询阶段使用同义词分析,搜索行为会是什么样的。在本例中,我们会生成相对于查询而言更为宽泛的匹配。就这个例子而言,这样的行为是否有意义是值得商榷的,但有些时候这样做可能是有用的。

如你所见,词汇在索引阶段的同义词匹配对应的是语义上等价或更为宽泛的值,这是我们期望的行为;而查询所匹配的,则是语义上等价或更为具体的文档。例如,对“apple”的查询,匹配的不只是带 apple 的文档,还有带 fuji 的文档。作为留给读者的一个练习,设想一下,如果我们只在查询阶段使用同义词分析,搜索行为会是什么样的。在本例中,我们会生成相对于查询而言更为宽泛的匹配。就这个例子而言,这样的行为是否有意义是值得商榷的,但有些时候这样做可能是有用的。

尽管利用同义词对专指性进行建模,好处是显而易见的,但它也不是一种可以被肆意滥用的模式。回忆一下 4.2 节查准率和查全率的较量,这种方法可以确保提高查全率,因为以这种方式返回的结果,包括了所有不包含同义词的文档,以及其他包含具体概念的文档。但不可避免的是,这种模式会给查准率带来问题。其中一个原因就是,这样做会令形式相同但语义不同的 token 有更多的机会被掺杂进来。例如, fuji 果真指的是 apple 吗?或者它是日本的一座城市呢?也许用户查寻“apples”想要的就是与这一层专指性相接近的文档,而不会考虑一篇专门讨论日本藤崎町的富士苹果栽培,和搜索相关到如此程度的文档。

相关性技术工程师还应该考虑到这种做法对  $TF \times IDF$  的影响,因为宽泛的词汇越是反复出现,其文档频率就越会被人为地提高,于是对宽泛的词汇形成的匹配就会被罚分。根据具体情况,这种处理方式对你而言也许是合理的,也许不是。也许把归一化和词频统计都关掉,然后使用一种固定常量的评价查询是值得一试的,这样一来对所有匹配的评价就都一视同仁了,不管其词频或文档频率是多少。这一类词汇的反复出现也可能会增加索引的规模。最后,使用这种方法需要搜集和维护一组额外的信息——同义词本身。

既然有这么多潜在的问题,还有什么理由选择这种模式呢?有的!对于利用分类体系来构造语义搜索的情况而言,这可能会是一个强大的工具。一个非常好的例子就是医学主题词表(Medical Subject Headings, 简称 MeSH),这是一个丰富的医学概念分类体系。假设我们的任务是构建一个医学著作的搜索引擎。文档根据其特定的 MeSH 主题被打上了标签。在索引阶段,同义词文件可以被用来匹配更为宽泛的主题。例如,假设我们对 MeSH 概念 depression(衰弱)、anxiety(焦虑)、

schizophrenia(分裂)进行了扩展,使其包含父一级类别:mental illnesses(精神疾病)。这样一来,就有可能在搜索如 schizophrenia 这样的子一级主题时,还能获得任何与精神疾病相关的文档。如果没有与 schizophrenia 相关的结果,搜索引擎就会转而在文档集合中寻找更为宽泛的概念,在本例中就是其他精神疾病的相关文档。

这样的补救办法看起来也许有点笨拙。不过试想一下,医学研究人员输入了多个搜索词。例如他们或许在寻找心脏疾病和精神分裂之间的联系,查的是“heart disease schizophrenia”。很遗憾,我们假设,他们并没有找到任何连接心脏疾病与精神分裂的直接匹配。相反,文档集合中包含了大量涉及心脏疾病的文档,其中有一部分提到了精神疾病。幸好,我们所采用的技术至少把这些非常接近的匹配都返回了。这些涉及心脏疾病和精神疾病的文章对于我们的研究人员而言也许是一个良好的开端,也许他们可以借助查到的这一点点心脏疾病与精神疾病的关联,激发起他们自己的灵感,以完善他们的研究。

将概念与各种不同程度的专指性映射起来是一项非常强大的技术,这是本节我们要阐述的理念。请记住这项技术,尤其当需要引导研究人员开拓创新的时候。下一节我们将为你展示另一种替代方法,同样也适用于分类体系。

#### 4.4.5 利用路径为专指性建模

在搜索技术里,同义词不是唯一能为专指性提供建模的方法。如果我们知道去哪里搜,专指性建模就会从众多常用的搜索模式中再次“蹦出来”。而它所依托的技术并非同义词,而是在查询期分析与索引期分析之间有意识的非对称策略。

让我们看一看涉及路径的常用搜索模式,以此来演示一下这项技术。假设我们正在构造一个文件系统的搜索引擎。作为其中一项功能,我们想让用户能够在指定的目录或子目录中寻找文档。例如,如果用户在搜索 /fruit/apples 目录中的文档,搜索结果也应该包含位于 /fruit/apples/fuji、/fruit/apples/gala、/fruit/apples/mcintosh 等目录中的文档。(是的,我们打算继续以水果为例!)

实现这一行为非常简单:

```
POST catalog
{ "settings": {
  "analysis": {
    "analyzer": {
```

```

    "path_hierarchy": {
      "tokenizer": "path_hierarchy"}}}},
"mappings": {
  "item": {
    "properties": {
      "inventory_dir": {
        "type": "string",
        "analyzer": "path_hierarchy"}}}}}}

```

首先我们设置了一个 path\_hierarchy 分析器，它将一个路径，比如 /fruit/apples/fuji 展开成若干个 token：/fruit、/fruit/apples，还有 /fruit/apples/fuji。然后我们将该分析器赋给字段 inventory\_dir，该字段代表了水果库存相关文件的存储目录：

```

PUT catalog/item/1
{ "inventory_dir":"/fruit/apples/fuji",
  "description":"crisp, sweet-flavored, long shelf-life"}
PUT catalog/item/2
{ "inventory_dir": "/fruit/apples/gala",
  "description ":"sweet, pleasant apple"}
PUT catalog/item/3
{ "inventory_dir": "/fruit",
  "description ":"edible, seed-bearing portion of plants"}

```

现在我们来搜一下 /fruit/apples/fuji 里的文件：

```

GET catalog/_search
{ "query": {
  "bool": {
    "should":
      [{"match": {"description": "<whatever>"}]}],
    "filter": [
      {"term": {"inventory_dir": "/fruit/apples/fuji"}}]}]}

```

我们得到了文件 fuji。如果在 /fruit/apples 中执行类似搜索，就会同时得到两个更为具体的有关苹果的文件，fuji 和 gala。最后，如果在 /fruit 下搜索，则全部三个文档都会被返回。（在有关分类体系的最后一节的讨论中，我们会看到类似的情况——此处我们所拥有的，是我们自己的一个小小的水果分类体系！）

也许你注意到了，inventory\_dir 配置并没有在索引阶段和查询阶段指定不同的分析器。因此乍一看似乎我们并没有使用非对称分析就已经在某种程度上为专指性做了建模。但是情况并非如此。在 Elasticsearch 里，词过滤器（term filter）是不做分析工作的；它的工作是查找与给定文本精确匹配的 token。因此，我们再一次看到了，对专指性的建模是通过非对称分析来完成的。

总结一下本节主旨，考虑下列几项规则：

- 如果索引中 token 所代表的是对查询阶段生成的 token 的一种泛化，那么搜索将会返回比搜索词更为具体的结果。
- 如果查询阶段生成的 token 所代表的是对索引中 token 的一种泛化，那么搜索将会返回比搜索词更为宽泛的结果。

在这一节中，我们了解了两种能够对专指性进行建模的方法，首先是同义词，然后是多级路径（hierarchical paths）。但是正如我们在讨论开始时所提到的那样，这种模式在整个搜索领域里随处可见。典型的例子——包括数字，甚至地理搜索！请大家接着往下读……

#### 4.4.6 对整个世界分词

到目前为止，本章讨论的焦点一直都是那些用于将文本转换成 token 的分析技术。但分析绝不仅限于文本。任何信息，只要包含可以被映射成离散语义单元的特征，都可以被表示成 token，存入倒排索引，并用于搜索之中。例如，这里有几类可被用于分词的信息，从简单到复杂：

- 数值型数据，包括整数、浮点数和日期。
- 地理信息，如经度 / 纬度坐标点或地理区域。
- 图像、形状、声音、质地……任何东西。

为了将搜索用于这些信息，我们要通过分析从中提取出有意义的、离散的特征。这就把搜索变成了一类用途更为广泛的系统，允许人们执行如图像搜索，甚至内容分类这样的任务。让我们来快速地看几个例子，这样，对于能被包含到搜索应用中的各种类型的信息，你就有一个大概的认识了。

#### 4.4.7 对整数分词

让我们来看一下数值型数据，特别是整数。能从整数中提取出来的最佳特征——对整数而言在语义上最有意义的符号——就是数字本身！不过搜索引擎对整数分词的方式乍一看也许有点出人意料。假设有一个搜索应用在逐年给历史事件建索引，比如，1945 年，第二次世界大战结束那一年。除了在建索引时为 1945 年生成一个 token，搜索引擎还会生成一系列 token，代表同一数字的各种低精度版本，如：



194、19 和 1。

可是为什么搜索引擎要这么做呢？这个嘛，试想一下，如果要对该字段做区间查询，我们需要做些什么呢？如果我们在建索引时天真地只为 1945 年生成了一个 token，那么为了查找落在区间内的那些年里的所有文档，我们就不得不遍历整个词典（term dictionary），找到每一个落入该区间的词，然后对那些匹配的文档（只要包含这些词中的任何一个）执行 SHOULD 布尔搜索。词的数量可能会数以百万计，因此这样做显然是无法实现规模化的（not scale）。不过，由于我们是在若干层次的精度上为年代建的索引，因此无须查看每一个词；我们可以利用这样一个事实，代表时间的低精度 token 涵盖了词汇的区间。

让我们利用上述认识，针对 1776 年至 2010 年间所有值得关注的事件，进行一次区间查询。在本例中，从 1776 年到 2010 年有 235 种可能的完整精度的 token，我们无须查看文档是否含有落在这一区间内的任何 token。如表 4.2 所示，通过在不同层次的精度上查询适当的词汇组合，我们就能找到相应的文档。

表 4.2 用于表现数值及其范围的 token

token	1776	1777	1778	1779	18	19	200	2010
涉及的范围	1776	1777	1778	1779	1800–1899	1900–1999	2000–2009	2010

将所有这些词汇都“或”在一起的这样一条查询会匹配一篇 1945 年的文档吗？会的，因为该文档在建索引时生成了低精度的 token，19，这正是搜索词中的一个。

这里还有一件事情要注意：搜索引擎为了有效执行数值区间查询而采用的这种技巧，是一种非对称分词技术。特别是，它在索引阶段对数字的分词要比在查询阶段更为宽泛。这正是我们此前讨论在搜索中为专指性进行编码时所介绍的模式。那时我们就说过，它还会再蹦出来的！

#### 4.4.8 对地理数据分词

地理搜索使用了一种类似的策略，不同之处在于其 token 是对二维信息进行编码，而非对线性的一维信息进行编码。也许看上去有点不可思议，搜索引擎竟然可以对二维信息进行编码。毕竟，搜索的核心是文本，而文本在本质上是线性的。但是除了生成 token 的流程外，什么也没有变；倒排索引和搜索算法依然以和之前一样的方式工作着。

假设有一个应用，正在为美国西部地区一些名字听起来很有趣或很奇怪的城市

建索引。一种在平面、矩形地图上为地理数据进行编码的简单方法，是采用 Z- 编码 (Z-encoding)。什么是 Z- 编码呢？通过图 4.6 来解释会更容易一些。

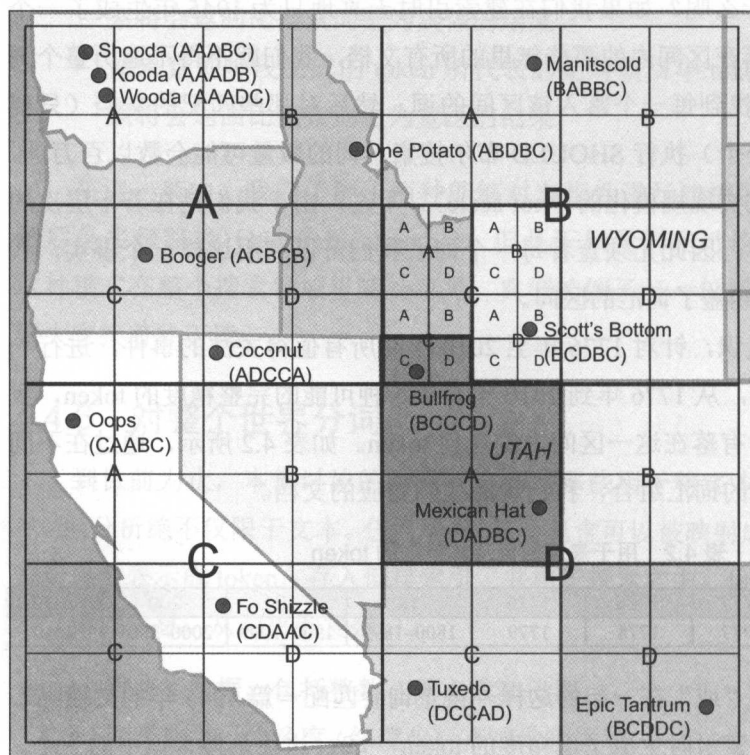


图 4.6 在一张矩形地图上经过 Z- 编码的几个坐标点的例子

该地图被分成 4 个象限，分别标以 A、B、C 和 D。每个象限被进一步分成了 4 个子象限，同样被标以 A、B、C 和 D。这一过程不断继续，直到地图被分成足够小的区域，达到期望的精度为止。注意，每一级象限的顺序都遵循 Z 字形，这就是其名称的由来。利用这种编码，地图上的每一个点都可以被翻译成一个由 A、B、C、D 组成的序列。例如，请把目光投向 Scott's Bottom（位于 Wyoming，即怀俄明州）。该处的 Z- 编码是 BCDBC，因为这座城市的顶层位于象限 B，接下来一层位于象限 C，再接下来一层位于 D，以此类推。

当为地理位置建索引时，我们首先找到位置点的 Z- 编码表示形式。然后，与为数字建索引的策略相似，在几个不同层次的精度上为 Z- 编码的坐标点建索引。例如，由 Scott's Bottom 生成的 token 会包含最高精度的位置编码，BCDBC，然后是较低精度的表示形式 BCDB、BCD、BC 以及 B。为了查到指定区域内包含的全部位置点，我

们需要找到代表地图上相应区域的那些词,将它们放到同样的 SHOULD 布尔查询中。例如,再次参见图 4.6, 查询 Utah (犹他州) 境内所有名字很有趣的市镇, 可以查 DA、BCCC 和 BCCD。参见上述地图, 我们可以看到 Bullfrog 和 Mexican Hat 都满足这一查询, 因为那两个城市有一个 token 前缀与查询 Utah 所用的查询词相匹配。如果要查 Wyoming 境内的城市, 你会选用什么查询词呢? 最后, 你是否注意到专指性模式再一次发挥了作用呢? 地理搜索实际上只不过是对一维数值区间查询所使用的模式在二维上的一种模仿。

#### 4.4.9 对歌曲分词

让我们以一个复杂一点的例子来结束本章的讨论, 一种可以被分词并纳入搜索的信息: 歌曲, 确切地说, 是用口哨吹出或用鼻子哼出的曲子。假设我们被要求建立一个搜索引擎, 允许人们用口哨吹曲子, 并搜索与之匹配的歌曲。

为了在这一过程中提供帮助, 应用开发团队的另一拨人已经建立了一个口哨编码器, 可以用它来接收来自手机麦克风的音频输入, 并对每一个用口哨吹出的音符, 判断其是否比上一个音符的音调更高、更低, 或保持不变。太棒了! 这就是我们以倒排索引中的 token 对口哨音符进行编码所需要的全部准备工作。

步骤如下。在被索引的曲子, 或者说用口哨吹出的旋律中, 每一个音符都有一个符号:

- 如果音符的音调比前一个高, 其符号为 U, 代表升调。
- 如果音符的音调比前一个低, 其符号为 D, 代表降调。
- 如果音符的音调与前一个相同, 其符号为 R, 代表重复。
- 第一个音符的符号为 \*, 表示这是一首曲子的开始。

这种标记法, 被称为旋律轮廓帕森斯码 (Parsons Code for Melodic Contours), 于 1975 年被开发, 旨在对歌曲进行编码和索引, 以使其能够被搜索。经典儿童歌曲 *Old MacDonald Had a Farm* 的开头部分可以被编码成 \*RRDURDURDRD, 如图 4.7 所示。

不过, 为了能把这些信息变成有用的、可以被搜到的 token, 我们还有更多工作要做。和以往一样, 我们需要识别出有意义的特征, 帮助区分不同的歌曲。当然, 我们可以把整首曲子的编码作为一个有意义的特征, 但是这样做有可能会带来一些潜在的问题。其一, 人们在哼整首曲子时有可能会弄错个别音符。要是我们的搜索

应用能容忍用户的错误就好了。把整首歌的编码作为一个特征这种做法的一个更大的问题是，尽管索引会保留整段编码，但是一个人会用口哨把整首歌都唱完的可能性几乎为零。用户有可能只会唱曲子里最熟悉的那一段。我们希望这样依然可以匹配索引中的曲子。

“♪♪” Old MacDonald had a farm E-I E-I O “♪♪”

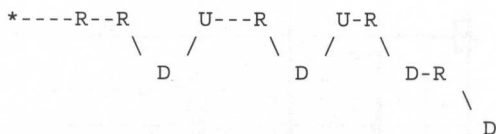


图 4.7 Old MacDonald Had a Farm 的帕森斯码

此处 N - 元组就可以派上用场了。一个 N - 元 token 过滤器每接收一个 token，都会将其拆分成若干更小的 token，代表原 token 的一个“加窗子集”（windowed subset）。例如，一个针对“Old MacDonald”的帕森斯码（\*RRDURDURDRD）的 5 - 元分词结果，如下所示：

[\*RRDU] [RRDUR] [RDURD] [DURDU] [URDUR] [RDURD] [DURDR] [URDRD]

前面两节，Elasticsearch 使用了专门的 Java 代码来为数值数据和地理数据进行分词。而在 Lucene 内部，正如前面几个简化了的例子所展示的那样，这是利用字节数组（byte arrays）而非字符数组（character arrays）来完成的。此处要想构建类似的功能就需要构造一个 Elasticsearch 的分析插件（记住，开源搜索框架是可以不断往上加插件的）。不过对于帕森斯码而言，如果我们坚持选择文本表示形式的话，那么就可以使用标准的文本分析技术了。在本例中，可按如下方式设置一个 N - 元分析器：

```
POST music
{ "settings": {
  "analysis": {
    "filter": {
      "parsons-ngram": {
        "type": "nGram",
        "min_gram": 5,
        "max_gram": 5}},
    "analyzer": {
      "parsons": {
        "tokenizer": "keyword",
        "filter": ["parsons-ngram"]}}}}}
```

此处我们所要做的全部工作就是新建一个名为 `parsons` 的分析器，其内部利用关键字分词器为输入数据生成一个 `token`，然后将该 `token` 传给 `N`-元 `token` 过滤器。对于 `N`-元处理流程的配置，元组的最小和最大长度均被设成了 5。此分析器可以采用对称方式加以使用。在索引阶段，一首完整歌曲的帕森斯码是 5-元的。（假设，歌曲的帕森斯码需要在此前单独生成。）然后在查询阶段，用户用口哨吹出或鼻子哼出的歌曲片段，也可以用同样的分析方法改编成 5-元帕森斯码。

这是一个将复杂信息用于搜索应用的很有意思的例子，除此以外，它还是一个绝佳的例子，告诉我们如何选择合适的特征来表示和区分索引中所存的数据。5-元帕森斯码让用户能够无须哼完整首曲子就能查找歌曲，哪怕他们唱的那一段有误，该项技术也依然至少会产生出一些 `token` 来匹配用户寻找的曲子。不仅如此，帕森斯码本身所编码的不过是相对音调的最基本概念——高的、低的或重复的。因此，哪怕用户无法唱出曲子来，也依然还是有一定希望找到他所要寻找的那首歌曲的。

但是我们可以做得更好。按照常理，匹配较长的 `N`-元组，应该要比匹配较短的 `N`-元组更有意义。因此我们可以将 `min_gram` 与 `max_gram` 的设置都增加到，比如，7。不过这还不够，因为如果用户没有匹配较长的 `N`-元组，我们依然希望能够至少将基于较短 `N`-元组匹配的最为相关的歌曲返回给用户。为此，让我们用变长 `N`-元组来建索引。我们可以试一下，将 `min_gram` 设成 4，`max_gram` 设成 7。那么，歌曲 *Yankee Doodle*，包含帕森斯码 `*RUUDUD`，将会生成如下 `token`：

```
[*RUU][*RUUD][*RUUDU][*RUUDUD][RUUD][RUUDU][RUUDUD][UUDU][UUDUD]
```

使用变长 `N`-元组还有一个好处，那就是曾在 4.3.1 节中介绍过的 `TF × IDF` 显然在这里就可以派上用场了。利用 `*`、`U`、`D` 和 `R` 组成的字母表，只有 108 种可能的 4-元 `token` ( $4 \times 3 \times 3 \times 3$ )，然而，采用类似的计算方法，会得到 2916 种可能的 7-元 `token`。大多数情况下，对 7-元 `token` 而言，其文档频率会更低一些。因此，由于其稀缺性，任何时候与 7-元 `token` 的匹配，相比于与 4-元 `token` 的匹配，都会得到更高的评价。读到这里，对于如何实现一个具有良好相关性用户体验的音乐搜索应用，我们已经有了一个大致轮廓了。

在继续后面的讨论之前，回忆一下前面三个例子——数值搜索、地理搜索，还有歌曲搜索。举这些例子的目的不是为了告诉大家如何具体实现这三个搜索应用。数值搜索和地理搜索早已内建于 `Elasticsearch` 中，我们也不太可能需要使用帕森斯

码对歌曲进行索引。列举这几个例子的目的是为了拓宽大家对搜索技术应用领域的认识。我们已经看到了搜索不仅限于单词和文本。搜索可以被扩展到任何领域，只要在这些领域中，能够提取到有区分度的特征，并以 token 的形式对其进行编码即可。

## 4.5 本章小结

- token 是对特征的一种表达，后者潜藏于文本，以及任何可以分词的数据中。
- 分析决定了 token 的构成，控制着查准率与查全率的权衡，这种权衡普遍存在于所有搜索结果集中。
- 分析将文本和数据转换成 token，预测用户如何在直觉上理解同一想法的不同表现形式（runs、running，以及 run 等价于 running 这一想法的不同表现形式）。
- 对查准率 / 查全率的矛盾，也许并没有我们想得那么严重。一种规避的方法是借助相关性排名。
- 对分析的把控，同时也控制着搜索引擎计算 TF 和 IDF 分值的方法。
- Lucene 中包含很多策略，能够帮助我们对各种形式的数据进行标准化处理。我们专门讨论了下列几种：
  - 以非空格符分割的文本（non-whitespace-delimited text）。
  - 以同义词捕获专指性的语义。
  - 基于路径和基于同义词的方法，用于捕获专指性：从含义宽泛的词汇（fruit）到含义狭隘的词汇（fuji apple）。
- 我们可以为位置信息、音乐曲目，还有许多其他类型的数据进行分词，将搜索引擎变成一种用途广泛、跨越各种数据类型的系统。



# 5 多字段搜索基础

## 本章要点

- 在搜索时满足多个用户目标
- 搜索文档中的多个字段以匹配用户搜索
- 将来自源数据的字段改造成搜索友好的格式
- 平衡不同搜索字段之间的彼此影响
- 理解多字段搜索策略的优缺点

之前我们曾将搜索与一本书的索引做过比较。这样的索引能让我们快速定位书页，找到我们关心的主题。假如你对法国大革命感兴趣，只要拿起一本介绍法国历史的书，翻到最后面，找到对应的书页即可。

类似的，搜索引擎利用倒排索引能够很快确定含有搜索词的文档。搜索单词“revolution”，搜索引擎就会列出一堆讨论革命的文档。在第4章中，我们的目标是利用分析手段来优化保存于倒排索引中的词汇，并最大化查准率和查全率。我们将内容的特征以 token 的形式表达，摒弃了 token 总是和单词相关的想法，而是将 token 与文档中包含的语义相关联。

如果老板拿着一个棘手的相关性问题对你穷追不舍，你得知道，除了对分析进

行优化以外，还是有很多办法能对搜索做出调整的！我们需要搞明白，搜索引擎是怎样广泛利用对业务至关重要的各种规则，对搜索结果进行排名的。文档几乎一定是由多个字段构成的。每个字段都有其自身的个性和特点。根据业务的需要，每个字段对搜索引擎的排名行为所产生的影响，也都有其自身的预期。

从本章开始，我们将逐渐采用一种自上而下的视角来对待搜索。搜索引擎不仅是书本后面的索引；它还是一个高度可扩展的内容排名系统，在表达业务（business）和用户（user）的优先级方面拥有强大的威力。在本章中，我们将着重于把多个字段逐次纳入排名方案之中，如图 5.1 所示。为此，我们将对多字段搜索（multifield search）的排名与评价进行剖析。每个多字段查询都有一个目的——这是一种能将各字段评价组合起来的有针对性的方法，它能平衡对业务和用户至关重要的各种规则。

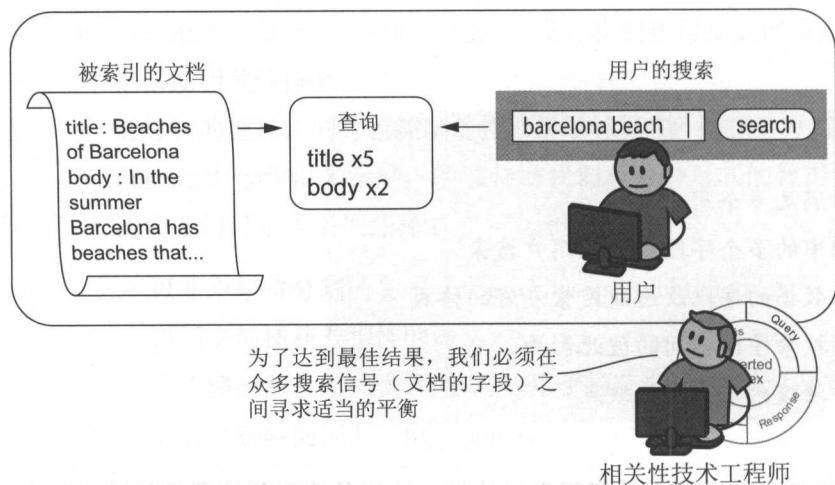


图 5.1 相关性技术工程师在排名时通过多个字段为排名规则指定优先级

为了掌握多字段搜索，我们还是会继续处理针对每个字段的匹配与评价。如果我们掌握了一些基本的情况，比如，标题或正文字段的相关度评价对用户的意义，则会有助于我们准确地把握方法，将这些字段组合起来，纳入一个更大的排名函数。在本章中，我们不仅要明白字段在搜索时的意义，还要学习构造字段的方法，目标明确地为整个搜索方案提供准确的信息。只有这样，我们才能有效地对排名函数进行编程，把对业务或用户至关重要的排名规则结合进来。

## 5.1 信号及信号建模

如何才能让搜索引擎懂得业务排名的规则呢？如何才能让搜索引擎的排名看起来像是一种可以通过编程手段得到的结果，而不是捉摸不定的东西呢？

首先，让我们来回忆一下那个甚至让经验丰富的搜索开发人员也时常感到神秘莫测、望而生畏的东西。在本书中，我们时常要盯着这些看似恐怖的相关性解释信息（relevancy explains），例如下面的片段：

```
2.555276, product of:  
3.1940954, sum of:  
3.1940954, weight(title:alien in 223)  
0.8, coord (4/5)
```

在这里要告诉大家的是：面对这个丑陋的大家伙，我们一定会成为驾驭它的高手！与其认为是高深莫测的搜索引擎捣鼓出来的这些评价数据，倒不如改变思路，想象每个数值都关联着一个信号——它是一种迹象（indication），表明了文档中存在着意义非凡或至关重要的东西。

### 5.1.1 什么是信号

我们所说的信号到底是什么意思呢？所谓信号（signal），是指任何参与相关性评价计算的组成部分，它对应于有意义、可度量的用户或业务信息。这里的用户或业务信息，可以是我们在排名过程中用到的任何信息，从“电影很卖座”到“影片简介与用户查询有关”再到“影片导演的名字在用户查询中被提到。”

也许我们并没有意识到，但是假如曾用多个字段做过搜索，其实我们早就已经开始在用信号来思考问题了。在第3章中，我们列举了在电影搜索中我们认为具备相关性的一系列字段，比如导演姓名，或影片简介之类的。通过搜索导演姓名字段，我们是在尝试对某个信号的强度进行度量，即“影片导演的名字在用户查询中被提到”，并提示搜索引擎可以在排名中利用这一事实。通过搜索正文部分的影片简介字段，我们同样是在试图搞清楚影片简介是否与用户的搜索字符串相关。让搜索引擎理解那些对用户、领域，或业务至关重要的排名规则，我们早已在摸索中向着这一方向在走了。

为了不只是简单地罗列字段，我们将在本章及后续几章中学习两项技术，这两项技术将让我们在创建信号并控制相关性排名时如虎添翼。究竟是哪两项技术呢？

首先，我们要掌握信号建模（signal modeling）的方法。不必把相关性评价当成什么深不可测的东西，是我们自己在掌控着它！在第4章中，在通过对特征的控制进行用户搜索建模时，我们已经小试牛刀。本章，我们将借助信号建模继续扩大战果。利用信号建模，我们将对字段的相关性评价进行全方位的把控，以此来度量什么是我们所需的。我们的目标是，不断寻找理想中的信号，对我们期望得到的信息做出更为准确的度量。

其次，我们还会使用排名函数（ranking function）。毕竟，本章的目的是介绍多字段搜索的。对多个字段的评价——即多个信号——需要被合并成一个总的相关性评价才行，这样才能平衡所有对用户或业务而言的重要因素。这就是排名函数的目标。我们将一些信号合并成更大的信号，以获得多方面的信息。在本章及后续几章中，我们将学习如何使用排名函数。后面两章主要讨论的是，Elasticsearch 中 multi\_match 查询的具体用法，以及与之相关的排名函数。再后面几章会告诉大家，如何进一步深入使用排名函数，对各种不同的信息进行放大、过滤，以及按优先级排序。

如果我们很清楚如何将用户和业务的规则转换成有意义的排名信号，那就可以用搜索引擎干很多事情。当我们懂得了如何利用字段来表达信号，我们就能营造出许多独特而又智能的搜索用户体验！

### 5.1.2 从源数据模型开始

不幸的是，在没有做信号建模时，多字段搜索时常会被搞得复杂不堪。人们并没有考虑字段是怎样被搜索的，以及评价时能提供什么样的信息，而是直接复制源数据的字段。所谓的源数据模型（source data model），是指存在于源系统中的数据结构，这样的系统可以是数据库、API，或者任何其他东西。尽管以这样的方式起步也还差强人意，但源数据模型并没有针对搜索做过优化。对它的优化是基于对源系统的考量而来的，比如某些数据库的或应用的需求。我们所得到的字段，其评价提供的信息是模棱两可的（ambiguous），这些字段无法在评价时给予我们具体而有针对性的信号。这样做往往会人为地使查询期间的逻辑变得晦涩难懂，导致多字段搜索既脆弱又复杂。

以公司的员工数据库为例。要将员工数据放入 Elasticsearch，一个显而易见的着眼点就是将员工属性从源数据库直接复制到 Elasticsearch 文档的相应字段中，如图 5.2 所示。如果员工数据库表有 first\_name、middle\_initial 和 last\_name

列，我们就将 `first_name`、`middle_initial` 和 `last_name` 这几个字段直接放入 Elasticsearch，而不考虑后面我们希望如何对其进行搜索。

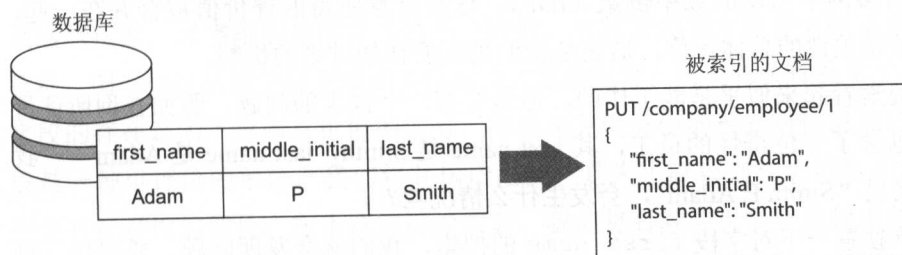


图 5.2 不加任何处理直接为源数据模型建索引

这样的字段结构给我们的多字段搜索带来了不少麻烦。例如，我们如何满足对姓名全称的搜索，在这种情况下，用户会以 `FirstName MiddleInitial LastName` 的格式（如 “Adam P. Smith”）在搜索框里输入搜索词。就目前的状况，我们能想到的最好的办法是，在每个可能的字段里逐一搜索每个词。如果我们使用了 `multi_match`，搜索引擎可能就会以清单 5.1 所示的方式来执行。

### 清单 5.1 脆弱的多字段人名搜索

```
usersSearch = "Adam P. Smith"
search = {
  "query": {
    "multi_match": { 1
      "query": usersSearch,
      "fields": ["first_name", "middle_initial", "last_name"],
    }
  }
}
```

用户的搜索词

将用户查询应用于字段 `first_name`、`middle_name`，以及 `last_name`

正如查询验证服务端（`query validation endpoint`）所揭示的那样（回想一下第 3 章中 Lucene 的查询语法），我们可以看到 `multi_match` 大概是怎样为每个搜索词逐一查询每个字段的：

```
(first_name:adam first_name:p first_name:smith) |
(middle_initial:adam middle_initial:p middle_initial:smith) |
(last_name:adam last_name:p last_name:smith)
```

这条搜索分别针对每个字段搜了 Adam、P 和 Smith。如何将其变成一个相关性的总体评价呢？每个字段针对各个搜索词都有其相应的评价。回忆一下前面几章，

针对给定搜索词在某字段中的  $TF \times IDF$  评价（如 `first_name:Adam`），其计算方法是，求得搜索词在文档的这一字段中出现的频率（TF），以及搜索词在这一字段的所有文档实例中出现的频率倒数（IDF）。这些计算所得的评价值被合并在一起，形成了一个相关性的总体评价，这正是我们稍后要详加讨论的机制。

前述搜索看起来似乎是能工作的，但是它有一个很大的问题。假如查询所评价的文档里包含了一位奇怪的员工，其 `first name` 是 Smith，`last name` 是 Adam——我们的会计老兄“Smith P. Adam”，会发生什么情况呢？

只要单独看一下对字段 `first_name` 的搜索，我们就会发现问题。要记住，每个搜索词针对 `first_name` 都会被搜一遍，即使这些搜索词代表的不是 `first name` 也是如此：

```
first_name:Adam first_name:P first_name:Smith
```

针对 Adam 在 `first_name` 字段里的搜索，其匹配对排名结果的影响力是很低的；而 Adam 又是一个普通的 `first name`，所以其文档频率很高。针对这一匹配的  $TF \times IDF$  评价就会相应偏低。我们在 `first_name` 里搜 Smith 的情况又会如何呢？好吧，只有一位员工的 `first name` 是 Smith——就是我们那位“Smith P. Adam”！Smith 是一个很少见的 `first name`，因此，针对 `first name` 是 Smith 的这种情况，其  $TF \times IDF$  评价就会相应偏高。一下子，出人意料的事情发生了，由于 `first_name` 中存在匹配 Smith 的情况，而 `last_name` 中又存在匹配 Adam 的类似情况，导致了搜索结果中 Smith P. Adam 远远地排在了 Adam P. Smith 的前面。

这里有一个关于多字段搜索的极大的反模式在起作用。我们没有根据这些字段被搜索的方式，仔细考虑它们提供的信号是什么。可以看到，利用 `multi_match`，我们针对每个搜索词，对每个字段都做了查询。在人名搜索中，这就导致了字段评价提供的信息变得模棱两可。比如，对 `first_name` 的评价，并没有提供有价值的信号，能反映搜索字符串中 `first name` 和文档中 `first_name` 的关系（能反映两者的关系也许会更 valuable）。相反，它提供的信息是模糊的。它假设任何搜索词都有可能是 `first name`，并给出了相应的评价。在构造排名函数时，这种做法帮助不大。

用信号建模构造出来的字段，查询时的二义性更小，它能像人一样，通过被搜索的字段读懂要解答的问题。在进行信号建模时，我们必须回答这两个问题：

1. 为了得到所需的信息，用户希望如何搜索这些字段？



## 2. 为了改进或调整这些信息，我们需要做哪些工作？

每个问题都千差万别。对人名的搜索和对餐馆或影评的搜索也不尽相同。要满足这些搜索，对字段的建模方式是不一样的。与其因为多字段搜索困难重重而心烦意乱，倒不如设法将字段变成相关性解决方案的财富。对于相关性技术工程师而言，字段的存在是为了返回有价值的信号，可以相关度分值的形式对信息加以度量。字段是一种可评价的单位，我们将其构造出来，是为了就查询与文档之间的相似度，产生有针对性的评价。在本章的后续部分，我们将看到各种多字段搜索的例子，以透彻理解这一概念。只有理解了字段并对其施以控制，我们才能构造出可以兼顾多个字段的相关性解决方案。

### 5.1.3 实现信号

让我们来看一个信号建模的小例子。回顾人名搜索，可以看到，信号建模也许会对我们所处理的问题有很大的帮助。如果我们要满足预想中的 `FirstName MiddleInitial LastName` 搜索，解决的办法并非只是对几个源数据模型里的字段用一下 `multi_match` 这么简单，我们要建立一个从源数据模型衍生出来的 `full_name` 字段，在搜索阶段为我们提供需要的信息，如图 5.3 所示。

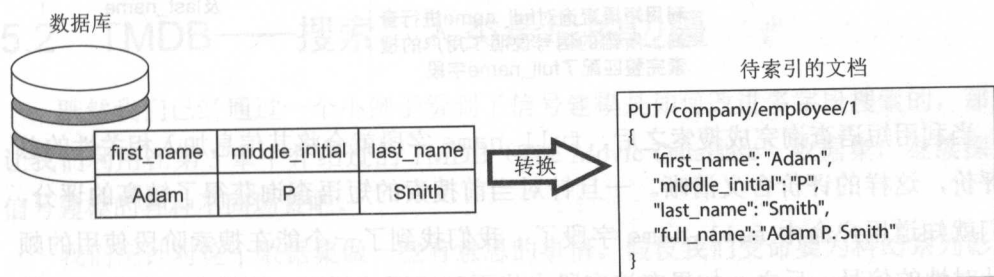


图 5.3 转换成 `full_name` 字段，能在排名时产生更好的信号。

如果我们知道用户一般是以 `FirstName MiddleInitial LastName` 这样的语法来进行搜索的，那么将名字以这种格式拼接起来，构造出一个衍生字段，这样的做法也许也是有效果的。一个多字段搜索的解决方案，除了单独搜索每个名字外，也许还可以针对这个 `full_name` 字段试着搜一下完整的全名。当用户搜的恰好是“Adam P. Smith”时，对 `full_name` 字段的匹配就可以和其他因素一起被放入排名函数，为用户返回准确的结果。也许排名函数看上去更应该像是这样的：

```
max(first_name:Adam first_name:P first_name:Smith),
  (middle_initial:Adam middle_initial:P middle_initial:Smith),
  (last_name:Adam last_name:P last_name:Smith))
+ full_name:"Adam P. Smith"
```

我们可以利用 Elasticsearch 的短语查询功能来实现这一思路，并利用 Elasticsearch 的布尔查询将结果合并起来。尽管这是我们第一次看到 Elasticsearch 的布尔查询，但是布尔搜索我们已经见过多次了。回想一下前面几章，我们利用 SHOULD 给包含搜索词的查询增加分值，这正是我们为创建所需的排名函数而要做的工作。改进后的多字段人名搜索如清单 5.2 所示。

清单 5.2 改进后的多字段人名搜索

```
search = {
  "query": {
    "bool": {
      "should": [
        "multi_match": {
          "query": usersSearch,
          "fields": ["first_name", "middle_initial", "last_name"],
        },
        "match_phrase": {
          "full_name": usersSearch
        }
      ]
    }
  }
}
```

利用多个SHOULD子句将两个评价组合在一起

用户的搜索词

将用户查询应用于字段first\_name、middle\_initial，以及last\_name

利用短语查询对full\_name进行查询，所给的信号说明了用户的搜索完整匹配了full\_name字段

当利用短语查询完成搜索之后，full\_name 字段就会将其信息加入相关性的总体评价，这样的评价含义清晰。一旦针对当前搜索的短语查询获得了较高的评分，我们就知道用户命中 full\_name 字段了；我们找到了一个能在搜索阶段使用的颇有针对性的信号。反之，如果在该字段中找不到短语的匹配，那就意味着情况截然相反，要么用户没有按 FirstName MiddleInitial LastName 这样的格式来搜索，要么没有匹配到用户的确切姓名。

#### 5.1.4 信号建模：为数据的相关性建模

如前面的例子所示，通过查询全名，我们超越了原来的数据模型，有针对性地解答了搜索用户的问题。在一些数据库系统中，数据建模是一项针对结构化数据的工作，其目的是为了在数据检索系统（data-retrieval system）的约束之下解答用户提

出的问题。我们对结构化数据进行数据建模，就是想凭借此类系统的优势，解决用户的问题。

对搜索引擎而言，情况亦是如此。只不过此处的信号建模是针对数据相关性所做的建模。作为一种数据检索系统，搜索引擎有其独到的优势。这种优势意味着，针对相关性的数据建模，其所采取的思路和源数据模型是不同的。搜索引擎对数据的处理秉承了索引优先（index-first）的理念。它允许文本被切分成一系列词，以使其有可能在倒排索引中被成功搜索到。大多数时候，这里的词对应的是单词。但正如我们在第4章所学到的那样，这不是必需的。我们所说的词不必对应于单词，或文本。

我们可以利用分词处理得到的词汇，回答用户的具体问题。凭借信号建模和前一章所学技术给我们带来的启示，我们构造出来的字段，能够充分利用搜索引擎的优势。对于如何对这些字段进行搜索，我们能够进行十分精细的控制。我们可以同时对查询和字段加以考察，设计出所需的相关性信号，为用户解决关键性问题。

如果我们能摒弃——字段的存在只是为了保存数据的属性——那样的观点，转而认同——为了让用户找到希望的数据，可以对其进行加工——这样的观点，那么我们就可以着手，将相关性原则通过有效的编程手段放入搜索引擎了。

## 5.2 TMDB——搜索，人类最后的边疆

既然我们已经通过一个小例子看到了信号建模是如何改进多字段搜索的，那就让我们利用在第3章中介绍过的TMDB（The Movie DataBase）数据集，继续探索信号建模的各种不同场景吧。

我们想针对这个数据集做一些有意思的事情。假设我们受命要为科幻系列影片“星际迷航（Star Trek）”的爱好者们打造一款电影搜索应用！我们已经大费周折地把数据从TMDB导入了搜索引擎。还有一个设计精美、充满科幻元素的用户界面，唤起了人们对“进取号”的美好记忆。剩下的唯一工作，就是连上搜索引擎，拿到搜索结果，然后大功告成！真是这样的吗？

我们即将看到，“星际迷航”的影迷们对相关性的定义有着他们自己的想法。不过，我们还得经历一次我们自己的迷航，穿越源数据模型，抵达相关性解决方案的彼岸，在那里，面对影迷的诉求，让方案为我们提供准确的信号。在本章及下一章，我们会运用在5.1节中学到的知识，建立排名函数，并构造信号，满足用户与业务

的需求。

在本章以及后续几章里，我们会将讨论的重点放在技术，而非流程上。也许这样做大家就会觉得可以让我们避免面对一些更难处理的问题了。（搜索引擎应该做些什么？如何找出用户的搜索需求？谁来定义这些需求？如何对搜索效果加以跟踪并持续测试，确保搜索尚未与我们的目标背道而驰？）关于这些更为高阶的议题，我们将在第10章做深入探讨。眼下，为了解决目标问题，更为重要的是要搞清楚与搜索引擎交互的一些细节：这是你作为一名研究相关性技术的工程师，每天都要做的技术工作。

在我们扬帆远航之前，先来回忆一下第3章中介绍的那些 Python 函数，那是我们在与 TMDB 数据以及 Elasticsearch 交互时用到的。在本章以及后续几章中，当与 TMDB 数据和 Elasticsearch 交互时，我们将用到这些函数，如表 5.1 所示。

表 5.1 与 TMDB 和 Elasticsearch 交互的基本操作

函数	描述
extract	返回一个字典，包含来自 <code>tmdb.json</code> 的影片从 ID 到详情的映射，反映了 TMDB 的源数据模型
reindex	让 Elasticsearch 重建索引，并传入：TMDB 的影片字典，针对分析的配置信息，以及字段的映射信息
search	给定 Elasticsearch Query DSL 的查询语句，对 TMDB 的 Elasticsearch 索引进行搜索

我们将对源数据模型中有关影片演职人员的信息进行考察。我们预计用户会搜索他们所喜爱的演员和导演。`tmdb.json` 中的每部影片都有 `movie['directors']` 和 `movie['cast']` 记录，包含了一组履历档案，是每位演员和导演的详细信息。如果了解我们究竟是如何利用 TMDB 的 API 将这些数据组合在一起的，请查阅附录 A。

回忆一下我们是如何使用前面那些辅助函数的，大家很容易就能想起在第3章中我们所做的工作。首先调用 `extract` 导入 TMDB 数据。然后调用 `reindex` 和 `search`，以英语作为默认的分析器对我们的文本字段进行分析，如清单 5.3 所示。

#### 清单 5.3 提取数据，建立索引，开始搜索

```
movieDict = extract()
analysis = {
    "analyzer" : {
        "default" : {
            "type" : "english"
        }}
reindex(settings=analysis,
        mappingSettings=None,
        movieDict=movieDict)
```

从 TMDB 中提取数据到 movieDict

默认使用英语分析器重建索引

```
usersSearch = 'basketball with cartoon aliens'
query = {
  "query": {
    "multi_match": {
      "query": usersSearch,
      "fields": ["title^0.1", "overview"],
    }}
}
search(query)
```

执行我们在第3章中建立的搜索

用户的查询

### 5.2.1 违反基本法则

“星际迷航”曾为人诟病的，是它有一套常被傲慢的星舰船长们违反的“基本法则”。那么，直接对 TMDB 建索引，同样也违反了我们先前给出的一些建议，我们直接把源数据模型放入了 Elasticsearch。难道不应该先做一下信号建模吗？如果直接利用这些数据来建立索引，难道就不会出现相关性问题了？

这个嘛，说得没错，不过我们那样做是有原因的。搜索是这样一种应用，它并不要求我们的优化尽善尽美。在我们还没有穷尽所有方向，找到一个完美的搜索解决方案之前，也许我们就已经到达了宇宙的“热寂状态”（heat death）。<sup>1</sup> 我们知道会有相关性存在问题，但并不十分清楚这些问题是什么，直到我们拿用户的搜索来做实验，问题才会显露端倪。很少有领域能像相关性搜索那样如此重视“快速试错（fail fast）”。我们载入数据，让系统基本可用，寻找问题所在，重新进行配置，依需重建索引，重新查询，梳理结果，如此往复。不断继续下去，直到我们得到了差距逐渐缩小的返回结果。

本章的目的就是要搞清楚源数据模型究竟在什么地方出的问题，在什么地方该花时间仔细研究信号建模。我们什么时候对特征进行建模，提取针对性、有价值的特征，然后放入词汇（terms）中？只有当搜索面对真实的用户查询时出了问题，我们才能做这些工作。

### 5.2.2 让嵌套文档扁平化

单纯从“工作原理”的角度而言，似乎对源数据模型进行加工这件事情不大可能。因为，我们为影片直接在 Elasticsearch 里所建的索引，其所面对的有可能是一种深度嵌套的 JSON 对象。而在本书的前面，我们讨论过 Lucene 的文档是一组扁平结构

<sup>1</sup> 此处意指决定解决方案的因素无穷无尽。——译者注

的字段。如何才能将一个层状结构的源数据模型映射到 Elasticsearch 里呢？为了理解如何搜索这样的数据，针对文档中的子对象来构造信号并对其进行遍历，我们需要搞懂 Elasticsearch 是如何处理层状结构的数据的。

让我们来看一看我们的老朋友——《空中大灌篮》，其文档在 Elasticsearch 里究竟是如何被存放的，参见清单 5.4。

清单 5.4 空中大灌篮的 Elasticsearch 文档片段

```
spaceJamId = 2300
httpResp = requests.get("http://localhost:9200/tmdb/movie/%s" % spaceJamId)
spaceJamDoc = json.loads(httpResp.text)
print json.dumps(spaceJamDoc['_source'], indent=True)
{
  ...
  "overview": "Michael Jordan agrees to help the Looney Tunes play
               a basketball game against alien slavers to determine their
               freedom.",
  "video": false,
  "id": 2300,
  "genres": [
    {
      "id": 16,
      "name": "Animation"
    }
  ],
  "title": "Space Jam",
  "tagline": "Get ready to jam.",
  "cast": [
    {
      "name": "Michael Jordan",
      "character": "Himself",
      "order": 0,
      "cast_id": 2,
      "credit_id": "52fe434bc3a36847f80496c9",
      "profile_path": "/7y16frD57Ztzk2mY4JeI2pQQhan.jpg",
      "id": 23678
    },
    ...
  ],
  "directors": [
    {
      "name": "Joe Pytko",
      "credit_id": "52fe434bc3a36847f80496c5",
      "job": "Director",
      "department": "Directing",
      "profile_path": "/c46Ah1KXlfc4W8mHVrGDsJ7dMPJ.jpg",
      "id": 23677
    }
  ]
}
```

输出

我们对第3章中的字段title和overview进行搜索

cast和directors字段

在上述文档中，有些字段对 Lucene 而言表现出了天然的扁平结构。同时我们也



看到了，`cast` 和 `directors` 字段对应的是一组人员信息。稍后，大家将会看到有关用户查找某人的各种不同用例，以及我们是如何进行信号建模，以满足相应的匹配规则的。

Elasticsearch 是如何对这些字段进行建模的呢？Elasticsearch 处理内嵌字段的方式怎样告诉我们信号建模是否成功呢？事实的真相是，Elasticsearch 给我们的字段加上了一点语法糖衣。它将这些对象转换成了多值的扁平化字段。每位演员的名字都被扁平化后放入了一个多值的 `cast.name` 字段，该字段包含多个人的名字。例如，下列内嵌对象：

```
"cast": [
  {
    "name": "Michael Jordan",
    "character": "Himself",
    ...
  },
  {
    "name": " Danny DeVito",
    "character": " Mr. Swackhammer (voice)",
    ...
  },
]
```

被成功转换成了多个并列的扁平化字段：

```
cast.name: ["Michael Jordan", "Danny DeVito", ...]
cast.character: ["Himself", "Mr. Swackhammer (voice)", ...]
```

或者从倒排索引的角度而言，单词 Michael、Jordan、Danny 和 DeVito 都存在于 Lucene 文档的 `cast.name` 字段中。因为 Michael 和 Jordan 都存在于同一个文档实例中，它们会作为两个紧邻的词一起被索引；对 Danny 和 DeVito 而言也是如此。而 `name` 或 `character` 字段的文本则看起来似乎应该是这样的：

```
cast.character: Himself BLAH BLAH BLAH Mr. Swackhammer (voice) BLAH BLAH...
cast.name: Michael Jordan BLAH BLAH ... BLAH Danny DeVito BLAH BLAH BLAH...
```

这使得我们在搜索人名时可以在查询中使用扁平化的 `cast.name`。

Elasticsearch 将这种表示形式称为内部对象 (inner objects)。其所维护的属性在搜索时颇具优势，但同时将字段扁平化以后，也失去了每个字段隶属于哪个子对象的关联信息。例如，这样的结构失去了 Danny DeVito 和 Mr. Swackhammer (voice) 之间的关联，即使它们被列在一起也无济于事。两个字符串都只不过是父文

档下属字段中的单词而已。

为了保留这种关联信息，Elasticsearch 还有其他方法对这样的关系进行建模。如果想进一步了解，请查阅 Elasticsearch 的文档中涉及内嵌文档（nested documents）和父子文档（parent-child documents）的部分，里面提供了一种涉及多文档关联的更有针对性的建模方法。<sup>1</sup> 目前阶段，对于信号建模工作，我们还是继续沿用默认的数据模型。

既然现在万事俱备，那我们就可以着手处理 TMDB，为“星际迷航”的影迷们解决相关性问题了。我们到底需要支持什么样的用例？源数据模型中的字段在什么情况下会给相关性带来麻烦？为了满足“星际迷航”粉丝们对电影搜索的需求，我们可能会采取什么样的信号建模呢？

## 5.3 在以字段为中心的搜索中给信号建模

让我们即刻开工，去满足“星际迷航”粉丝们对电影的好奇心吧！如前所述，我们的开发同事已经为“星际迷航”的粉丝门户网站提交了一个光鲜靓丽、充满科幻元素的搜索界面。我们受命让搜索能够工作。毕竟，一个看似光鲜却没有相关性搜索结果的用户界面，拿什么来引导用户体验呢？

为了打造一款搜索解决方案，返回的内容能让“星际迷航”的粉丝认为是有相关性的，我们必须要考虑对他们而言什么样的排名规则才有可能是最重要的。这些规则向我们预示了可能的排名信号。当然，像电影片名和简介这样的简单匹配依然是很重要的。此外，“星际迷航”的粉丝们还对他们所喜爱的演员情有独钟。因此，对演员的匹配可能会是一条重要的规则。由于很多演员通常也是“星际”系列电影的导演，所以搜索导演同样也是一条重要的排名规则。

以上这些因素也许并不十分准确，但是没关系。对于构造能够用来解决用户问题的信号和排名函数而言，从这些因素出发是很合理的。在我们对解决方案的测试过程中，一定还会发现更多的规则。一开始的时候，我们会发现在源数据模型中，有些字段产生出来的信号，也许能在搜索时直接反映上述排名规则：

- title——较高的评价可能相当于直接匹配标题。

---

<sup>1</sup> 更多信息，请查阅Elastic的博客，*Managing Relations*，位于[www.elastic.co/blog/managing-relations-inside-elasticsearch](http://www.elastic.co/blog/managing-relations-inside-elasticsearch)。

- overview——较高的评价相当于一条描述影片内容的查询。
- cast.name——较高的评价可能相当于匹配演员。
- directors.name——较高的评价可能相当于匹配导演。

我们还没做过任何信号建模，因此目前所产生的信号似乎还不太理想。但是我们需要一个初始方案作为起点。我们应该从什么样的排名函数入手，才有可能建立起一个合乎情理的总体评价呢？有哪些方案可供选择呢？在这一节里，我们将开始探讨排名函数和信号之间的相互作用。如何利用最基本的多字段搜索能力来建立排名函数？这些排名函数是如何利用我们所提供的信号的？如何在所选排名函数的上下文中，优化对字段的评价，改进搜索结果的质量呢？

对多字段排名函数的选择，决定了搜索结果以何种方式来呈现。它控制着搜索结果的整体排名方式。例如，我们的搜索是否应该由所有选项中被认为最重要的那个信号来决定？又或者，搜索在排名时应该平等看待多方因素吗？

让我们来看一下多字段搜索都有哪些选择。基于 Lucene 的搜索应用有两种通用的方法可以为多字段进行排名，如图 5.4 所示。

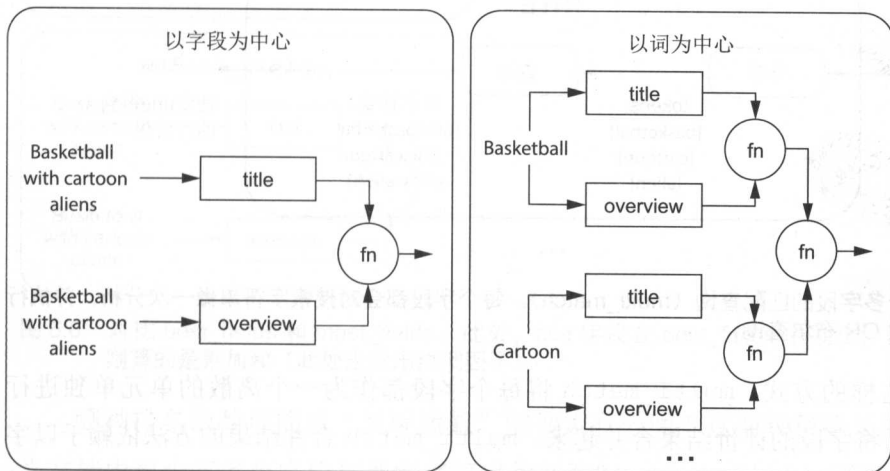


图 5.4 以字段为中心和以词为中心的搜索所呈现的 title 和 overview 字段。前者每个字段单独搜索；后者每个字段逐词搜索。

以字段为中心（field-centric）的搜索针对每个字段执行字符串搜索，待每个字段单独搜索完毕后，再将评价结果合并。以词为中心（term-centric）的搜索则刚好相反，它对字段的搜索是以逐词方式进行的。其结果是每一个词都有各自的评价，该评价是将词在每个字段中的影响合并以后得到的。作为对“星际迷航”搜索的首

次尝试, 我们将从以字段为中心的排名函数入手。以字段为中心的查询时常是分析的入手点, 因为这能让我们把重点单独放到每个字段以及搜索该字段所得到的信号上。我们将在第6章继续讨论以词为中心的搜索方法。

Elasticsearch 将以字段为中心的选项放入 `multi_match` 查询。我们已经多次见过 `multi_match`, 但是它究竟是怎样工作的呢? `multi_match` 会对每个传入的字段执行搜索。针对每个字段, 它会为搜索字符串在查询期执行分析, 并在得到的 token 上执行布尔搜索, 每个 token 都对应一个 `SHOULD` 查询子句。换言之, 对于一个使用英语分析器的给定字段而言, 搜索字符串会经历如图 5.5 所示的流程。

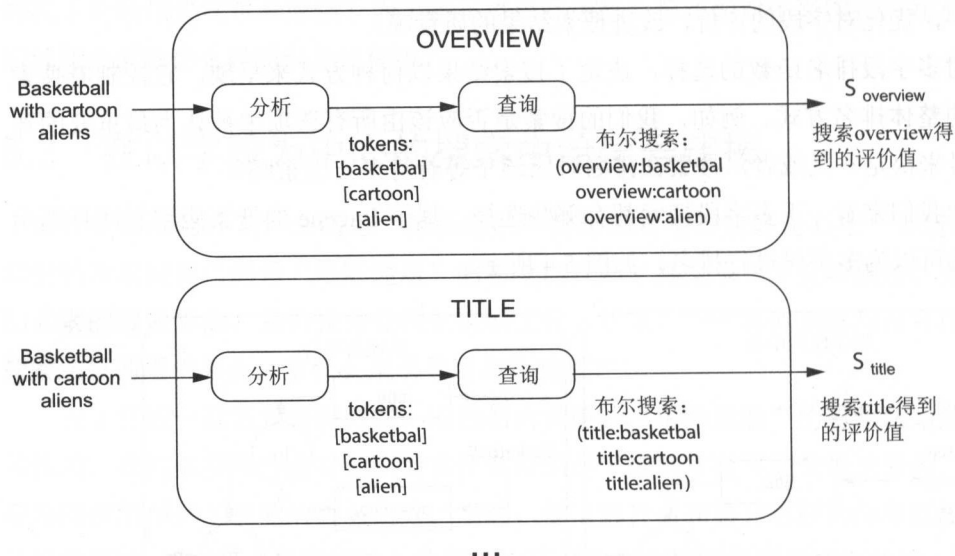


图 5.5 基于多字段的匹配查询 (`multi_match`)。每个字段都会对搜索字符串做一次分析, 并执行一次 OR 布尔查询。

通过这样的方式, `multi_match` 将每个字段都作为一个离散的单元单独进行搜索, 然后将字段的评价结果合并起来。`multi_match` 合并结果的方法依赖于以字段为中心的搜索的具体类型。让我们来看一下有哪些可用的排名函数。如果我们以  $S_{\text{overview}}$  代表单字段搜索 (`overview:basketbal overview:cartoon...`) 得到的评价结果, 那么对于以字段为中心的搜索, 我们会看到两种主要的形式可供选择。

- `best_fields`——默认情况下, 选取评价最高的字段。如果指定了 `tie_breaker` 参数 (取值从 0 到 1), 则对其余字段的评价会被放入整体评价之中。

假设 `title` 的评价最高，其计算方法如下所示：

$$\text{score} = S_{\text{title}} + \text{tie\_breaker} \times (S_{\text{overview}} + S_{\text{cast.name}} + S_{\text{directors.name}})$$

- `most_fields`——将每个匹配的评价都看作布尔查询中的一个条件子句。回想一下，布尔查询是一个带协调因子或称之为 `coord` 的加和结果。`coord` 等于匹配的条件子句数 / 总的条件子句数。因此，匹配的条件子句越多，`coord` 给予布尔查询的奖励也就越大：

$$\text{score} = (S_{\text{overview}} + S_{\text{title}} + S_{\text{cast.name}} + S_{\text{directors.name}}) \times \text{coord}$$

图 5.6 形象地展示了上述情况。

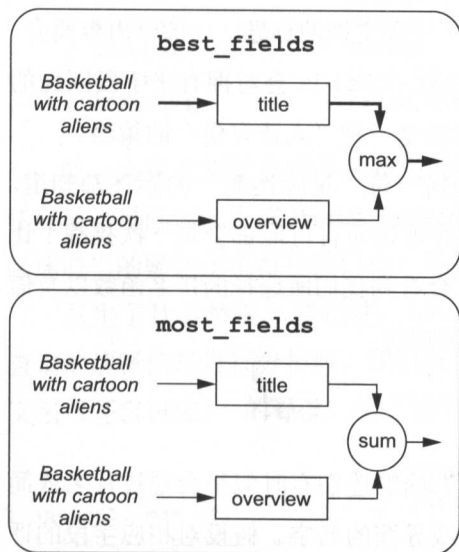


图 5.6 对比 `best_fields` 和 `most_fields`。此处，`title` 字段在 `best_fields` 中胜出，而 `most_fields` 则算是累加和（此处未给出协调因子）。

哪种排名函数可能对“星际迷航”的搜索有效呢？这两种策略是彼此对立的。当文档中很少有多多个字段与搜索字符串相匹配时，`best_fields` 策略更有效。这让我们更容易选择“最佳字段（best field）”。而 `most_fields` 方法则相反，当预计文档中有多个字段与搜索字符串相匹配时，这种方法更有效。

通过选择与搜索相匹配的评价最高的字段，`best_fields` 认定，“本次搜索必定与该字段有关。”如果我们的信号建模是合理的，每个字段的评价都是一个信号，那么 `best_fields` 就会选择最为合适的信号作为评价结果。在某种意义上，

`best_fields` 就像是在做一次职业面试——为搜索词挑选最为优秀的字段。它最终是想为每篇文档都建这样一个信号，告诉它们，“这次搜的是标题，而不是别的！”`best_fields` 的策略可以被称为是一种“胜者为王”(winner-takes-all)的搜索。胜出字段的评价结果会被采纳，而所有其他“紧随其后”者都将被忽略或低估——即被解释成错误的或低优先级的。

在 `most_fields` 里，各个字段就像是在进行团队合作。信号越多，表明相关性越大。这种方法归根结底就是“协作”！我们利用 `most_fields` 声明一组字段，所有这些字段都会被纳入统计，使相关性评价结果得到提升。搜索词与文档中匹配的字段越多，对文档的评价结果就越高，而匹配字段越少则文档评价越低。`most_fields` 是在表达，“理想中的搜索字符串包含了一篇文档的标题，一部分内容简介，再加上一位演员的名字。”通过这样的方式，`most_fields` 会对拥有多字段匹配的文档给予更高的评价。`most_fields` 可以被称为是一种“人皆有份”的策略。

无论怎样，所有这些都只是理论罢了！任何一位“星际迷航”的影迷都知道，深空探索对于任何一门有关宇宙运行机制的优秀理论而言可能都会是一次挑战！让我们赶紧来看一看我们的 TMDb 搜索吧，大家会看到我们所选择的排名函数以及信号建模方式是如何对返回相关性搜索结果产生影响的。

### 5.3.1 从 `best_fields` 开始

回顾一下我们的问题，看起来“星际迷航”的影迷们有时似乎会通过片名或简介来搜索某部片子，而有时候他们则会搜演员或导演的名字。假设对相应字段的评价与上述这些信号一一对应，我们的搜索似乎在倾向于这样一种策略：要么采纳某个字段的评价，要么采纳另一个的，而其他字段都一概忽略。这会不会是 `best_fields` 的一种用例呢？

为了试一下 `best_fields` 查询，让我们来搜一下“星际迷航”的演员“Patrick Stewart”，看看结果是否如我们预计的那样。我们预计 `best_fields` 可能会明白我们搜的是一位演员，并选择相应的字段作为最佳字段，参见清单 5.5。



清单 5.5 “星际迷航”的查询，素材取自第 3 章

```
usersSearch = "patrick stewart"
```

```
query = {
```

```
  "query": {
```

```
    "multi_match": {
```

```
      "query": usersSearch,
```

```
      "fields": ["title", "overview",
```

```
                  "cast.name", "directors.name"],
```

```
      "type": "best_fields"
```

```
    }
```

```
  }
```

```
}
```

```
search(query)
```

```
Num  Relevance Score
```

```
1    0.5308861
```

```
2    0.5308861
```

```
3    0.5308861
```

```
4    0.5308861
```

```
5    0.42397094
```

```
Movie Title
```

```
Legion
```

```
Halo 4: Forward Unto Dawn
```

```
Priest
```

```
Dark Skies
```

```
Drive Angry
```

用户的搜索

字段及放大形式取自第3章（加入了演职人员的姓名）

搜索结果

哇，大家立刻就发现了，我们的第一次尝试，表现十分糟糕！搜索结果里没有任何与“星际迷航”或 Patrick Stewart 有关的影片。如果这样的系统发布出去，“星际迷航”的影迷还不得挤破门框！还是赶快装上光子鱼雷吧！

发生了什么情况？要记住，best\_fields 只选择胜出的字段。当对这些搜索查询的解释信息进行分析时，我们发现系统对一位名叫 Stewart 的，影片 *Legion*（中文名《基督再临》）的导演，给了特别高的评价：

```
1.3460261, max of:
```

```
  1.3460261, product of:
```

```
    2.6920521, sum of:
```

```
      2.6920521, weight(directors.name:stewart in 868)
```

```
    0.5, coord(1/2)
```

取best\_fields各项的最大值

针对字段  
directors.name的布尔查询（协调因子 x 各匹配项之和）

匹配  
directors.name  
字段

对照一下另一部影片的解释信息，由演员 Patrick Stewart 主演的 *Star Trek: Generations*（中文名《星际迷航：星空骑兵》）：

```
0.38644278, max of:
```

```
  0.38644278, sum of:
```

```
    0.14300151, weight(cast.name:patrick in 533) [PerFieldSimilarity],  
    result of:
```

```
    0.24344127, weight(cast.name:stewart in 533) [PerFieldSimilarity], result  
    of:
```

取best\_fields各项的最大值

期望匹配  
Patrick Stewart

对导演 Stewart 的匹配，其评价似乎远高于对相应的 `cast.name` Stewart 的匹配。排在最前面的结果匹配的是导演，而非演员。这与直觉相悖。为什么与导演的匹配会给予这么高的评价呢？我们搜的是演员，而非导演，难道这一点不应该是一目了然的吗？搜索结果有失偏颇，如图 5.7 所示。

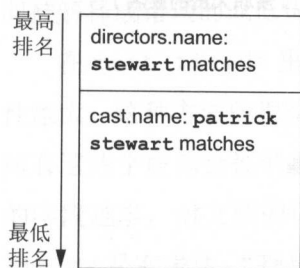


图 5.7 导演在很多匹配中都被选为最佳字段，而演员 Patrick Stewart 的匹配则评价较低，被排在了末尾。

从根本上说，我们并没有发挥出 `best_fields` 的优势。包含人名的搜索字符串既有可能匹配 `directors.name` 字段，也有可能匹配 `cast.name` 字段。如果我们没有通过放大处理（`boosting`）明确给出优先级，`best_fields` 带来的效果看上去就像是在洗牌一样，它将某个字段的匹配抽出来放到了顶端。如果没有我们的帮助，这样的洗牌不会太理想，也不会太直观。这里有几个原因。

首先，对字段的评价不是稳定的、线性可比较的。对于每个  $TF \times IDF$  评价而言，并不存在像从 0 到 100 这样绝对的相关性标尺，此处 0 表示完全不相关，100 则表示完全相关。因此，我们无法对两个前后相继的字段评价进行比较。我们真正可以比较的，只有像两个 `cast.name` 字段的评价这样的情况。如果我们考虑一下评价的工作原理，就会明白这是为什么了。词频、文档长度，还有倒置文档频率，所有这些度量在不同字段间的分布是不同的。这些因素都被计入了对字段的评价之中，于是就导致了这样一种结果，2.0 对 `directors.name` 而言也许是一个糟糕的评价，而 0.2 对 `cast.name` 而言却有可能是一个不错的评价。这两个评价是完全没有可比性的！从这个意义上而言，`best_fields` 的这种择“优（最大值）”机制并不是那么理想。这就像是在某人以英尺为单位的身高和以米为单位的身高之间选择“最大值”一样！这两种度量系统是无法相互比较的，除非我们通过数学手段让它们变得可以互相比较。

其次，也许是更为重要的， $TF \times IDF$  评价针对搜索者可能会搜索的内容有很大的倾向性。请记住， $TF \times IDF$  评价非常倾向于稀缺的词（ $IDF$  与稀缺性紧密相关）。但用户搜索的，更有可能是那些普通常见的事物。如果我们去一家食品杂货店购买咖啡，可能会更乐于被带到售卖咖啡的过道处，那里的咖啡品种繁多。要是被带到售卖冰激凌的过道，也许就不会那么开心了，那里只有几桶咖啡冰激凌等着我们。

同样的道理，单词 `stewart` 对于一名演员而言十分常见，但对于一名导演而言却很少见。由于这个原因，用户搜索演员而非导演的可能性更大。然而  $TF \times IDF$  却背道而驰，它更看重稀缺性。因此对这位潜力股导演的评价，要远远高于更有可能成为候选者的那位人尽皆知、却太过常见的演员。

换言之，我们得到的结果可能是有倾向性的——常常不是按照我们所期望的倾向来。这样可能会产生让用户困惑的搜索结果，因为鲜为人知的字段匹配比众所周知的字段匹配更为重要。

### 5.3.2 控制搜索结果中的字段偏好

在 5.3.1 小节中，`best_fields` 产生的搜索结果有失偏颇。它首先列出的是一位导演的匹配，然后才是一名演员的，以此类推。看似随意的倾向性让人觉得有违直觉（为什么导演会比演员更重要呢）。不过，有意而为的倾向性也许只是我们搜索应用的一张入场券而已！<sup>1</sup> 如果我们认为至关重要是将包含演员的影片放到搜索结果的最前面，而其他匹配次之，那又该怎么办呢？

利用放大处理，我们可以让 `best_fields` 向着我们期望的一侧倾斜。例如，假设我们对 `directors.name` 实施 `down-boost` 处理，就能降低它相对于其他匹配的优先级，如清单 5.6 所示。这样得到的结果，包含 Patrick Stewart 主演的影片就会更多，而字段 `director` 的优先级则会降低。此处，回想一下我们无法对字段评价进行比较的这一事实，因此按比缩小到原来的 0.1 并不意味着“重要性减少 10 倍”。放大系数只不过是一个经验倍数，用来确保我们的倾向性是符合预期的。

---

<sup>1</sup> 此处意指决定解决方案的因素无穷无尽。——译者注

清单 5.6 降低 directors.name 的影响

```

usersSearch = "patrick stewart"
query = {
  "query": {
    "multi_match": {
      "query": usersSearch,
      "fields": ["title", "overview",
        "cast.name", "directors.name^0.1"],
    }
  }
}
search(query)

```

用户的查询

将directors.name缩小为原来的0.1

Num	Relevance Score	Movie Title
1	0.46373135	Vertigo
2	0.46373135	Star Trek: Insurrection
3	0.46373135	Gnomeo & Juliet
4	0.46373135	Star Trek: First Contact
5	0.46373135	Excalibur

搜索结果

Patrick或Stewart的影片

现在我们的搜索倾向已经从导演转向了演员，因此搜索结果看上去好多了。其结果似乎匹配了名字里有 Patrick 或 Stewart 的演员，如图 5.8 所示。

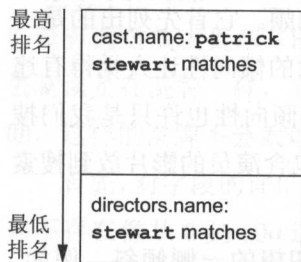


图 5.8 利用放大系数使错误的导演匹配排名降低

当我们希望构造一个有倾向性的排名时，应该使用 `best_fields`，由某个字段得到的结果占据绝对优势，而由其他字段得到的结果则排在后面。当有不止一个字段存在匹配时，我们需要通过放大处理明确指出哪个字段应当优先。在前面的例子中，`directors.name` 的影响低于 `cast.name`；这样做使搜索结果变得上下颠倒，我们将位于末尾的结果“拉”（drag）了上来，而位于开头的结果则“推”（push）了下去。

不过还是有一个问题。例如，由 Jimmy Stewart 主演的 *Vertigo*（中文名《迷魂记》），匹配了我们搜索中的“Stewart”部分。而该片另一位演员的名字是 Patrick，与“Patrick”相匹配。因此我们所度量的并不完全就是我们想要的结果。与

`cast.name` 字段评价相关联的信号并未真实反映出针对演员 Patrick Stewart 的匹配。相反，我们所得到的匹配是单独针对搜索词“Patrick”和“Stewart”的。因此，尽管我们为了满足需求，已经修改了排名函数，但是被度量的信息并不精准。是时候改进信号了！

### 5.3.3 可以使用信号更精准的 `best_fields` 吗

虽然我们已经将 `best_fields` 按照我们设想的倾向性颠倒了过来，但这依然没有得到我们的目标结果。字段 `cast.name` 有许多错误的匹配，因为有一些名字叫 Stewart 和 Patrick 的其他人也出现在了影片中。回想一下 `multi_match` 对字段执行的 `SHOULD` 查询 (`cast.name:patrick cast.name:stewart`)。我们还没有建立起这样一个概念，要求这些单词应该像一个整体一样结合在一起。如果我们能够创建一个字段，可以更确切地指出名字是否匹配，那么我们就能够极大地提高搜索的准确性。这样我们就有机会去掉很多错误的结果了。

如果我们想对一个搜索词中出现的名字和一位演员是否匹配做出度量，需要对字段进行某种信号建模。如果我们不是单独匹配像 Stewart 或 Patrick 这样的单词，而是设计一个信号模型实现更为准确的匹配，这样的思路如何呢？如果我们能够强制某个字段仅当精确匹配 Patrick Stewart 时才给予评价值，这样的做法如何呢？那样的话，就会产生一个准确的信号，用数字告诉我们，“搜索与 Patrick Stewart 是相匹配的”。

在第 4 章中，我们讨论过词汇专指性的程度问题。或者换个说法，匹配一个词的难度有多大？我们设计的匹配有多严格？如果字段要完成匹配有非常大的障碍需要克服（例如，需要精确匹配 Patrick Stewart 而不只是匹配 Stewart），那我们就知道该字段的精确性已经得到了很大的提高。这样的精确性失去了名字以变体形式匹配的可能性，有可能会降低查全率。但所幸的是，电影演员几乎一定都是以 `FirstName LastName` 这样的形式被引用的（“Patrick Stewart”而非“Stewart Patrick”）。因此在这种情况下，我们也许不必匹配名字的变体。

让我们增加一个信号来改善 `best_fields` 的查询效果。`shingle token` 过滤器能够根据由两个单词构成的词组来生成 `token`。它可以帮助我们构造字段，匹配由两个单词组成的人名。如果我们构造的分析器主要借助英语分析器和 `shingle` 处理，那就产生一种被称为短语索引（`phrase index`）的东西。一个短语索引利用由两个或三个单词构成的词组作为词汇（`terms`）。以短语 Patrick Stewart Runs 为例，提

取到的 token 类似 Patrick Stewart 和 Stewart Runs。请记住，token 和被索引的词不必非得是单个的单词！在本例中，我们已经将词映射为由两个单词构成的词组，或者称为双联词（bigram）。索引类似如下所示：

```
field cast.name.bigrammed
term patrick stewart
  doc 0
    freq 1
    position 1
  doc 2
    ...
```

当 Elasticsearch 对搜索字符串进行分析之后，我们得到了一个表示“相等”的分析结果，如图 5.9 所示。

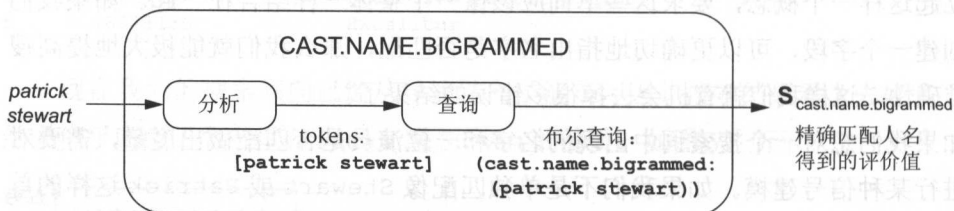


图 5.9 搜索 cast.name.bigrammed，该字段用于对双联词进行索引。对这样的字段进行搜索会得到一个比直接逐词匹配更为准确的信号。

清单 5.7 修改了对 TMDb 索引的分析方法。该清单配置了一个能生成双联词的 shingle 过滤器，叫作 bigram\_filter。使用该过滤器的同时，我们还创建了一个与英语分析器执行同样步骤的 english\_bigrams 分析器，只不过它最终生成的是双联词，而非一个个单词。一旦配置完毕，我们就可以在字段映射中使用该分析器了。

#### 清单 5.7 提取英语双联词的分析方法

```
analysisSettings = {
  "analyzer" : {
    "default" : {
      "type" : "english"
    },
    "english_bigrams": {
      "type": "custom",
      "tokenizer": "standard",
      "filter": [
        "standard",
        "lowercase",
        "porter_stem",
        "bigram_filter"
      ]
    }
  }
}
```

定制英语分析器以使其产生双联词



```

    "bigram_filter"
  ]
}
},
"filter": {
  "bigram_filter": {
    "type": "shingle",
    "max_shingle_size": 2,
    "min_shingle_size": 2,
    "output_unigrams": "false"
  }
}
}
}
}

```

利用bigram\_filter实现双联词的生成

我们用来生成英语双联词的bigram\_filter

接下来,我们要对索引和搜索使用上述分析器。利用 Elasticsearch 的多字段功能,我们可以在许多字段上生成双联词版本的 token。Elasticsearch 的这一特性让我们得可以在一个字段上同时进行两种形式的分析。清单 5.8 在为我们提供了经过普通英语分析之后的 cast.name 字段外, 还提供了一个相应的 cast.name.bigrammed 字段。(我们将对导演字段采取同样的策略, 此处不再赘述。)

清单 5.8 对 \*.bigrammed 字段做映射, 并重建索引

```

mappingSettings = {
  "movie": {
    "properties": {
      "cast": {
        "properties": {
          "name": {
            "type": "string",
            "analyzer": "english",
            "fields": {
              "bigrammed": {
                "type": "string",
                "analyzer": "english_bigrams"
              }
            }
          }
        }
      }
    }
  }
}
reindex(analysisSettings, mappingSettings, movieDict)

```

现在, 当 best\_fields 选择评价时, 我们的双联词字段应该会提供更有区分度的评价结果。基于这样的分析规则, 对字段的搜索会采用两个单词构成的短语。一条针对 “Patrick Stewart” 的搜索, 将试图精确匹配词语 [patrick stewart]。精确包含 Patrick Stewart 的文档同样也会在其 \*.bigrammed 字段里包含该词。让我们再次执行搜索, 这次加上双联词字段, 如清单 5.9 所示。

清单 5.9 搜索 \*.bigrammed 字段

```

usersSearch = "patrick stewart"
query = {
  "query": {
    "multi_match": {
      "query": usersSearch,
      "fields": ["title", "overview",
                 "cast.name.bigrammed", "directors.name.bigrammed"],
    }
  }
}
search(query)

```

用户的查询

搜索人名字段的双联词版本，而不是非双联词版本

搜索结果

Num	Relevance Score	Movie Title
1	0.7239306	Star Trek: Insurrection
2	0.7239306	Gnomeo & Juliet
3	0.7239306	Star Trek: First Contact
4	0.7239306	Excalibur
5	0.6334393	Conspiracy Theory

这是一个十分显著的进步！所有这些电影都是由 Patrick Stewart 主演的。通过增加一个可供排名函数使用的专门的信号，我们已经极大地改善了搜索结果的准确性。如果我们采用 `best_fields`，应该就会得到准确的信号——因为这样的信号极为准确，所以搜索字符串不太可能会匹配大量的文档。尽管如此，依然存在多个字段同时匹配的可能性。因此我们需要定义对信号的放大处理，明确哪些字段是我们认为最重要的。即使是这些双联词字段也可能需要给予放大处理，以表明对演员的匹配优于对导演的匹配。例如，演员 Jonathan Frakes，同样也执导过许多“星际迷航”的电影。当存在冲突时，作为相关性技术工程师的我们，需要通过对信号的放大处理明确哪个字段是最重要的。在本例中，我们可能依然希望确保对演员的匹配优于对导演的匹配。

### 5.3.4 让失败者分享荣耀：为 `best_fields` 校准

像“Patrick Stewart”这样的搜索，其所搜索的内容很明显要么属于范围 A 要么属于范围 B。`best_fields` 很适用于这样的场景，因为只有一小部分字段会匹配成功，并将演员 Patrick Stewart 的匹配在搜索结果中的排名推高。要是用户搜索的是多项内容，比如“Star Trek Patrick Stewart”，那又该怎么办呢？在这个例子中，搜索者指定了两条规则：电影 Star Trek 和演员 Patrick Stewart。规则之一，“Star Trek”，非常适合匹配 `title` 字段；而“Patrick Stewart”则非常适合匹配 `cast.name`。用户想

要在搜索中使用多条规则，希望搜索时系统将所有规则都考虑在内，这种情况是很常见的。

要同时考虑多条用户搜索规则，纯粹的 `best_fields` 方式无法满足要求。除了“搜索该名演员的目的是为了……”这样的信号以外，我们也想将其他信号包含进来，比如“搜索这部影片的目的是为了……”。然而 `best_fields` 关注的是某个字段相对于所有其他字段的相关性评价。如果对片名的评价更高，那么对 Patrick Stewart 的匹配就会被 `best_fields` 彻底忽略掉。这样的搜索开始令 `best_fields` 优势不再。

如果我们想让 `best_fields` 继续工作，但同时也希望能兼顾重要性次之的字段，那该怎么办呢？例如，搜索“星际迷航”的用户也许会在排名时将针对片名的匹配作为首要信号。其他规则，包括对演员姓名、导演姓名，或影片描述的匹配，则应次之。这样，我们就建立起了一种先主后次（`first-second sort`）的排序方法——即优先考虑 `best_fields` 选中的信号，但允许其他信号在该最优字段范围内扮演一定的角色。这就是 `tie_breaker` 所要做的工作。回忆一下 `tie_breaker`，它允许我们对没有在搜索结果中胜出的匹配，从中挑出一些来计入评价结果：

$$\text{score} = S_{\text{title}} + \text{tie\_breaker} \times (S_{\text{overview}} + S_{\text{cast.name}} + S_{\text{directors.name}})$$

上述方案产生的搜索结果如图 5.10 所示。

Star Trek matches	Patrick Stewart matches
	Non-Patrick Stewart matches
Non-Star Trek matches	

图 5.10 一种先主后次的排序方案，`best_fields` 仍将胜出，但在胜出字段的范围内，对其他规则的匹配允许文档排名获得提升。

要达到先主后次的效果，要求我们对字段的 `boost`（放大系数）和 `tie_breaker` 取值进行仔细调整。要记住，`boost` 值并不等同于优先级。对 TMDb 而言，事实上对 `title` 的评价是很高的（正如我们在第 3 章看到的那样）。因此我们需要对 `cast.name.bigrammed` 相对较小的基础评价分进行放大，从而将其提升至仅次于 `title`

的地位。在下列查询中, 针对 `cast.name.bigrammed` 的 `boost` 值为 5, 而 `tie_breaker` 为 0.4, 给 `best_fields` 带来了预料之中的竞争关系: `title` 第一, 而 `cast.name.bigrammed` 第二:

```
usersSearch = "star trek patrick stewart"
query = {
  "query": {
    "multi_match": {
      "query": usersSearch,
      "fields": ["title", "overview",
        "cast.name.bigrammed^5", "directors.name.bigrammed"],
      "type": "best_fields",
      "tie_breaker": 0.4
    }
  }
}
search(query)
```

用户的查询

将 `cast.name.bigrammed` 放大5倍

将 `tie_breaker` 设为 0.4 用以计入其他字段的评价值

Num	Relevance Score	Movie Title
1	0.35363546	Star Trek: Insurrection
2	0.35363546	Star Trek: First Contact
3	0.34679613	Star Trek: Generations
4	0.34285474	Star Trek: Nemesis
5	0.33423716	Star Trek

搜索结果

使用 `best_fields` 会优先选择匹配片名的信号, 同时加入一部分从匹配人名的信号中获得的评价。最后剩下的全部都是“星际迷航”的影片。在这一组搜索结果中, 有一个次要的信号, 它从根本上指出了影片是否由 **Patrick Stewart** 主演。满足要求的结果被移到了这组“星际迷航”影片的开头位置。而明显在开头处缺失, 且向着搜索结果的末尾处渐渐式微, 最终消失在视线之外的, 则是那些非“星际迷航”的影片, 或者 **Patrick Stewart** 未曾参演的影片。

而 `tie_breaker` 则会让 `best_fields` 的搜索偏离非此即彼的极端。它允许我们在评价中对其他信号进行考量。事实上, 如果将 `tie_breaker` 设成 1.0, 我们就会得到所有字段评价的加和:

$$\text{score} = S_{\text{title}} + 1.0 \times (S_{\text{overview}} + S_{\text{cast.name}} + S_{\text{directors.name}})$$

这样做就得到了如下结果:

$$\text{score} = S_{\text{title}} + S_{\text{overview}} + S_{\text{cast.name}} + S_{\text{directors.name}}$$

这种做法看上去有点像 `most_fields` 中的加和处理。你会发现这种处理形式也许是很有诱惑力的。假如我们想要的不是带有倾向性的结果, 或是先主后次的排

序行为，假如我们想让每个字段对总体评价都有所贡献，那该怎么办呢？

### 5.3.5 利用 `most_fields` 统计多个信号

`best_fields` 是一个“薄情”的家伙！我们已经讨论过如何利用它来选择最优匹配。凭借准确的信号，`best_fields` 能够判断出我们的搜索与哪个字段相匹配。通过在众多信号中选择其一而放弃其他，我们可以利用这种方法来确定，“本次搜索搜的是人吗？”或者“本次搜索搜的是影片吗？”这很大程度上取决于在对搜索规则进行度量时我们所使用信号的准确程度。同时，很大程度上也取决于利用放大处理为字段建立优先顺序时我们有多大的把握——切记字段评价中存在的那些奇怪而矛盾的情况。

当用户指定多个搜索条件时，比如“Star Trek Patrick Stewart”或“Star Trek Patrick Stewart William Shatner”，搜索就更像是多个信号聚合以后的结果了。如果我们选择强调某个信号，比如“本次搜索涉及的是电影片名”，我们就会丢失用户告诉我们的其他搜索规则。借助 `best_fields` 的 `tie_breaker` 参数，我们找到了能够解决这一问题的一种办法。利用 `best_fields` 和 `tie_breaker` 来满足这样的用例，这种做法看上去更像是 `most_fields` 的行为。记住，`most_fields` 会在每个底层字段上执行一次布尔查询。其效果相当于底层信号的加和：

```
score = (Stitle + Soverview + Scast.name + Sdirectors.name) × coord
```

将 `most_fields` 看作一组 `SHOULD` 布尔子句，有助于我们正确理解使用这项技术的方法；这些 `SHOULD` 子句根据每个字段的对应信号，列出了文档相关性的所有匹配规则：

- 搜索字符串应当（`SHOULD`）提及电影片名。
- 搜索字符串应当（`SHOULD`）提及影片简介中的文字。
- 搜索字符串应当（`SHOULD`）提及影片演员的名字。
- 搜索字符串应当（`SHOULD`）提及影片导演的名字。

理想的文档会全部满足上述 4 点需求。与电影片名、影片简介、演员名字和导演名字都匹配的搜索字符串会跳到排名结果的开头。与其中 3 个信号相匹配的搜索字符串则紧随其后，以此类推，如图 5.11 所示。

像 `best_fields` 那样，把焦点只放在一个字段上的日子一去不复返了。相反，

我们要让每个字段都在最终的评价结果中拥有发言权。让我们来看一下，如果选择 `most_fields` 来处理 “Star Trek Patrick Stewart” 搜索，会发生什么情况，如清单 5.10 所示。

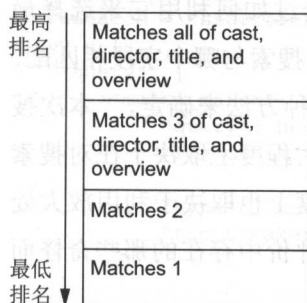


图 5.11 `most_fields` 很大程度上倾向于具有更多匹配字段的搜索

#### 清单 5.10 搜索 Star Trek Patrick Stewart

```
usersSearch = "star trek patrick stewart"
query = {
  "query": {
    "multi_match": {
      "query": usersSearch,
      "fields": ["title", "overview",
        "cast.name.bigrammed", "directors.name.bigrammed"],
      "type": "most_fields"
    }
  }
}
search(query)
```

← 用户的查询

← 改成使用 `most_fields`

← 搜索结果

Num	Relevance Score	Movie Title
1	0.57795894	Star Trek: Generations
2	0.37984636	Star Trek: Insurrection
3	0.37984636	Star Trek: First Contact
4	0.37325242	Star Trek: Nemesis
5	0.20443419	Star Trek

与带有 `tie_breaker` 的 `best_fields` 类似，这种方法会让 Patrick Stewart 主演的“星际迷航”电影排名靠前，这正是我们想要的结果。为了让满足匹配规则的文档排名靠前，我们用上了所有的信号。在本例中，这让理想中的文档浮出了水面——一部由 Patrick Stewart 主演的“星际迷航”电影。

当用户指定多个匹配规则时，将多个匹配信号组合在一起，这样的排名方法往往会与用户的期望更加吻合。`best_fields` 要想获得带有倾向性的结果就需要付



出很大的努力，而 `most_fields` 则将所有给定的字段评价都结合在了一起。当用户期望的不是提高某个字段的优先级，而是想得到一个由多字段构成的有针对性的集合时，那么 `most_fields` 就是最好的选择。在 `best_fields` 中，理想的文档只会匹配一个字段而不管别的字段。在前面的例子里，我们提高了片名的重要性，以此来主导评价。而 `most_fields` 的策略则并不迷恋于某个字段的评价，相反，它让每个字段的评价都有话语权。当用户就匹配哪个字段并没有一个明确的“目标”，而是列出了许多具有同等重要性的匹配规则时，`most_fields` 就是正解。

### 5.3.6 在 `most_fields` 中缩放信号

虽然 `most_fields` 试图给出与规则最为匹配的结果，但字段评价并不总是那么配合。要记住，某个字段的评价也许天然就是要比其他字段高出一个量级，没有什么特别的理由。由于评价之间的巨大差异，我们得到的搜索结果也许看起来更像是带有倾向性的 `best_fields`，而非期望中的 `most_fields`，也就是那种“每个字段对评价都有贡献”的行为。如果将前面搜索得到的结果进一步向下展开，我们预计先主后次的搜索结果中所包含的影片，应该既是 Patrick Stewart 主演，又是“星际迷航”的片名。很遗憾，结果并非如此：

6	0.16354734	Star Trek: The Motion Picture
7	0.16354734	Star Trek Into Darkness
8	0.14310393	Star Trek VI: The Undiscovered Country
9	0.14310393	Star Trek V: The Final Frontier
10	0.14310393	Star Trek IV: The Voyage Home
11	0.14310393	Star Trek II: The Wrath of Khan
12	0.14310393	Star Trek III: The Search for Spock
13	0.10484329	Maps to the Stars
14	0.086285345	Star Wars: The Clone Wars
15	0.06411133	Star Wars: Episode VI - Return of the Jedi

这些全部都是片名匹配 Star Trek 或 Star 的影片！为什么会这样？不管什么原因，对片名字段的评价似乎再一次胜过了对非片名字段的评价。针对 Star Trek VI（中文名《星际迷航6：未来之城》）的片名评价如下所示：

```
0.5196225, weight(title:star in 281)
```

相比之下，在评分上可供比较的一部最为接近的 Patrick Stewart 影片是 *Ted*（中文名《泰迪熊》）：

```
0.19781886, weight(cast.name.bigrammed:patrick stewart in 831)
```

考虑一下与 `most_fields` 相关的加和：

```
score = (Stitle + Soverview + Scast.name + Sdirectors.name) × coord
```

要记住，对字段的评价其实是不可比较的。尽管凭借 `coord` 值，我们对多字段匹配有着极大的倾向性，可我们还是需要通过适当地 `boost` 值来得到一个能带给我们均衡表现的公式。在使用 `most_fields` 的同时，我们通过放大处理来平衡这个具有明显倾向性的公式：

```
score = (Stitle + Soverview + Scast.name + Sdirectors.name) × coord
```

举个例子，假如对 `title` 做 `down-boost` 处理，或者对 `cast.name` 做 `up-boost` 处理，我们也许就有可能获得与目标更为接近的结果，如清单 5.11 所示。

#### 清单 5.11 对 `title` 做 `down-boost` 处理

```
usersSearch = "star trek patrick stewart"
query = {
  "query": {
    "multi_match": {
      "query": usersSearch,
      "fields": ["title^0.2", "overview",
        "cast.name.bigrammed", "directors.name.bigrammed"],
      "type": "most_fields"
    }
  },
}
search(query)
```

用户的查询

降低title的权重以平衡其评价带来的影响

最终得到的搜索结果，除了 Patrick Stewart 主演的“星际迷航”电影外，看起来更像是“星际迷航”电影与 Patrick Stewart 参演电影的一个混合：

6	0.04985989	Ted
7	0.047840547	Star Trek
8	0.044039465	The Beaver
9	0.038272437	Star Trek: The Motion Picture
10	0.038272437	Star Trek Into Darkness

这种做法的挑战在于，我们需要仔细调整 `boost` 值，以便确保 `most_fields` 能够兑现其承诺。否则，如果随意扩大某个字段的评价，我们最终会得到预料之外的倾向性结果。`best_fields` 中的放大处理明确了优先级，即哪个字段的匹配在预期的倾向性中应该被优先考虑，而 `most_fields` 中的放大处理则给词的加和带来

了均衡，从而使参与评估的这些字段能够回到一种更为均衡的评价结果上来。

### 5.3.7 什么时候其他匹配才无关紧要

在结束对 `most_fields` 的讨论之前，来看一下我们可能会遇到的 `most_fields` 的“阿基里斯脚踝”。不过我们在这里只是阐述问题，至于问题的解决方案则会在第 6 章中给出。正如许多“星际迷航”的影片中所呈现的那样，我们也将为你带来一点扣人心弦的情节。

`most_fields` 策略会把满足所有匹配规则的文档放到最前面。这看起来是一件非常明智的事情。可是很多时候，我们不应该在同时具备两个较强信号的情况下强化文档的相关度。例如，考虑下面这条搜索：

```
usersSearch = "star trek patrick stewart william shatner"
query = {
  "query": {
    "multi_match": {
      "query": usersSearch,
      "fields": ["title", "overview",
                 "cast.name.bigrammed", "directors.name.bigrammed"],
      "type": "most_fields"
    }
  }
}
search(query)
```

用户的  
查询

如果我们了解一点“星际迷航”影迷们的情况，也许会猜到针对“Star Trek Patrick Stewart William Shatner”的查询可能是在寻找一部由 Patrick Stewart 和 William Shatner 共同主演的“星际迷航”影片。`most_fields` 给出的结果真是这样的吗？这个嘛，多半可能是如下所示的情况：

Num	Relevance Score	Movie Title
1	0.5415871	Star Trek V: The Final Frontier
2	0.39785004	Star Trek: Generations
3	0.35108924	Star Trek IV: The Voyage Home
4	0.3037074	Star Trek: Nemesis
5	0.19542062	Star Trek: Insurrection

为什么一部只有 William Shatner 主演的“星际迷航”影片（*Star Trek V*，中文名《星际迷航 5：终极先锋》），其排名会高于一部由 William Shatner 和 Patrick Stewart 共同主演的影片（*Star Trek: Generations*）？这是因为 William Shatner 同时主演并执导了 *Star Trek V: The Final Frontier*。要记住，`most_fields` 所刻画理想文档是这样的：

- 应当 (SHOULD) 匹配搜索字符串中出现的片名。
- 应当 (SHOULD) 匹配搜索字符串中出现的导演。
- 应当 (SHOULD) 提及搜索字符串中出现的演员。

对 *Star Trek V* 而言，导演和演员都有匹配，更加满足理想文档的条件。而在 *Star Trek: Generations* 中，有两位演员获得匹配，这增加了该信号的强度。

最终的结果是，我们并没有非常准确地刻画出理想中文档的样子。上面列出的信号并没有与用户所设想的相关性排名保持一致。要使用 `most_fields`，就要仔细考虑什么是“理想的文档”。前面列出的匹配规则是否合适呢？也许更好的规则应该是这样的：

- 应当 (SHOULD) 匹配搜索字符串中出现的片名。
- 应当 (SHOULD) 匹配任何一个搜索字符串中出现的影片相关人员。

相比于专门针对“导演”或“演员”来构造信号，一个更为适合的信号是“在搜索字符串中出现的某人是否与电影有所关联？”也许我们的搜索用户并不关心 William Shatner 是一名导演还是演员，只要他与影片有关联就行。

以词为中心的搜索 (term-centric search)，正如我们将在第6章中看到的那样，为搜索提供了一种更加自上而下的视角。以词为中心的搜索有助于我们更进一步地回答用户所提出的较高层面 (high-level) 的问题，而这些问题也许与源数据模型相去甚远。

### 5.3.8 有关 `most_fields` 的结论是什么

`most_fields` 为我们提供了一种指定理想文档的方法。但我们在使用时务必小心。如果放大处理和信号使用得当，针对特定的搜索，`most_fields` 往往能够获得更好的解决方案。如果我们不是很确信，比如说，究竟哪个信号应当被优先处理，那么 `most_fields` 就会是一个很好的切入点。但是我们应该避免在使用 `most_fields` 时的一种反模式：为增加信号而增加信号。很多时候，多字段匹配与相关度的增加并不是紧密相关的。

## 5.4 本章小结

- 用户并不关心我们的数据库是如何存储数据的。他们需要的是数据以利于搜索的形式呈现出来。
- 信号的作用是将相关性评价与有意义的排名规则建立起对应关系（餐馆的距离要近，搜索的是电影片名，等等）。
- 利用信号建模，我们构造出来的字段能更好地映射对用户有意义的匹配规则。
- 利用排名函数，我们将信号组合在一起，以得到搜索结果的总体评价。
- 以字段为中心的搜索会将整个查询字符串用于每个字段，最后才把针对各字段的评价结果合并在一起。
- `best_fields` 对搜索采取了一种“胜者为王”的态度；拥有最高评价的字段会被选中作为最终的评价结果。
- `most_fields` 对搜索采取了一种“人皆有份”的态度，将所有字段的评价放在一起求和。
- `best_fields` 的 `tie_breaker` 参数允许我们加入来自其他字段评价的影响，使得 `best_fields` 更有点像 `most_fields`。

# 6 以词为中心的搜索

## 本章要点

- 考察以字段为中心的搜索为什么不能满足用户的朴素预期
- 探讨哪些源数据模型会令搜索用户感到困惑
- 通过对比来阐述各种以词为中心的搜索方法的优缺点
- 解释以词为中心和以字段为中心的两搜索方法之间的矛盾
- 将用户对搜索的朴素预期与更高明的搜索功能结合起来

第5章我们介绍了信号（signals）和多字段搜索（multifield search）。信号是用来对诸如“搜索是标题的精确匹配吗？”或者“搜索中提到某位演员或导演了吗？”这样的查询条件进行度量的。这类信号依赖于我们控制查询和构造字段，根据用户意图进行建模的能力。我们称这一过程为信号建模。只有当字段与信号清晰对应起来之后，我们才能在多字段搜索的策略中着手对其加以权衡。



前面的章节主要讲述以字段作为相关性的核心单位。但是用户并不关心字段，他们认为搜索词才是搜索的核心组成部分。用户并不想介入数据库或应用程序的细节之中，他们期望能够在短时间内找到自己满意的搜索结果，期望我们能够满足其对搜索的简单诉求。因此，以词为中心（term-centric）的搜索与其他形式的多字段搜索有所不同，它更关注的是搜索词，而不是内容的结构。

在本章中，我们将讲述以词为中心的搜索技术，如图 6.1 所示。相比于其他因素，此类搜索技术更加关注用户的搜索词——其关注的程度，甚至可能会高于我们认为理应考虑的那些因素！我们会看到，以词为中心的搜索往往能够完胜我们在前面几章精心设计的各种字段搜索。前面所讲到的，通过对字段构造和分析过程加以控制而设计出来的强有力的信号，将会被一种更加广义的相关性概念所取代，它强调用户对文档结构更为简单的理解。然而用户永远是一个矛盾体；如果非要他们做出选择的话，他们还是会优先考虑搜索中有关位置(locations)、观点(ideas)和人员(people)的智能匹配。所以事实上，对于以词为中心的搜索所体现出来的朴素行为，必须借助我们在前面几章中设计出来的诸多智能信号，对其加以平衡和增强才行。

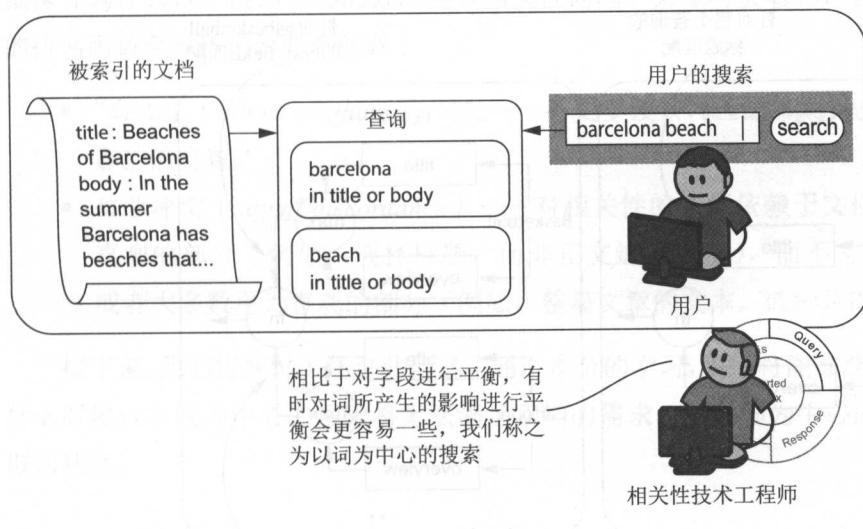


图 6.1 以词为中心的搜索更加关注用户的搜索词

## 6.1 什么是以为中心的搜索

我们来回忆一下，以字段为中心的搜索是将整个搜索字符串放入每个字段内进行

行搜索，然后再将所有字段的评价值结合起来，以此实现对多个字段的搜索。每个字段都有自己的特殊使命，用以精确度量重要的业务或用户信息——例如：“影片导演在搜索查询中被提到了吗？”或者“用户正在搜索影片的片名吗？”

以词为中心的搜索从另一个不同的角度展开搜索。图 6.2 阐明了该方法的不同之处。以词为中心的搜索在处理搜索字符串时就像一个逐词（term-by-term）匹配器，它查找的是每个搜索词（search term）的理想匹配，而非将整个搜索字符串在每个字段中都去搜一遍。因为每个搜索词的理想匹配项可能存在于不同的字段中，所以最终的相关性评价值是几个字段匹配混合之后的结果。一条针对“basketball cartoon”的搜索，可能会在文档的标题（title）字段中匹配 basketball，而在文档的正文（body）字段中匹配 cartoon，从而得到一个逐词计算的多字段混合分值。因此，以词为中心的搜索更为关注用户指定的查询条件，而较少关注某个具体的字段。用户并不在乎什么字段，他们只是希望自己的搜索词能够有所匹配！匹配什么都行！毕竟，如果每个搜索词都能找到它们各自的理想匹配，那我们不就达到了搜索相关性的理想境界了吗？

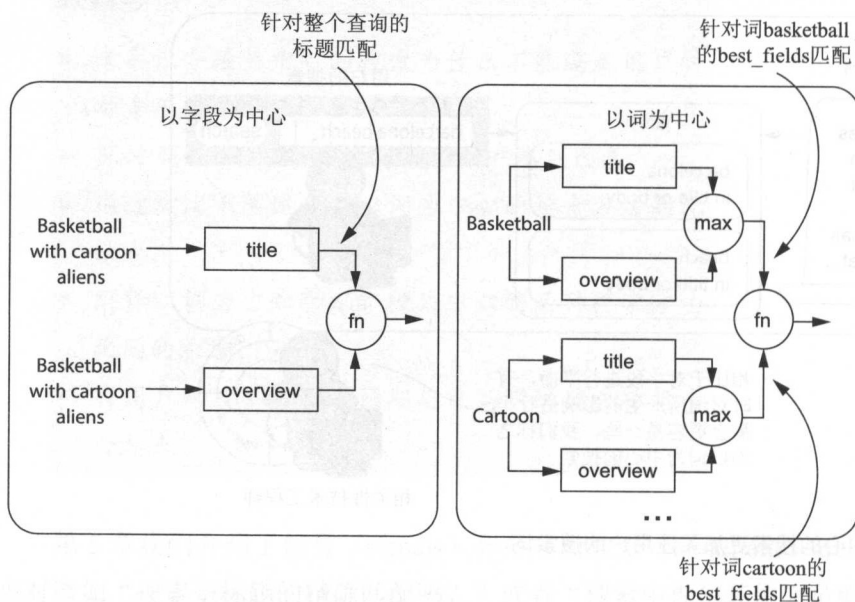


图 6.2 以词为中心的搜索将每个搜索词放到一组字段集合中，最后产生的评价混合了文档中各字段以逐词方式计算得到的评价。

如果真是这么简单就好了！在实际使用中，以词为中心的搜索，其所承诺的好

处略带有一点欺骗性。用户觉得自己并不关心字段，然而实际上他们仍然会期望搜索能够智能地匹配与文档相关的各种属性，譬如：人员（people）、位置（locations）、标签（tags）和观点（ideas）。虽然用户并没有意识到这一点，但是他们在很多地方仍然需要以字段为中心的搜索，以词为中心的方法和以字段为中心的方法构成了一种针对搜索的非常重要的“阴阳互补”。正如大家在本章中即将看到的那样，每一种搜索都有其自身的局限和优势。相关性的艺术就是要博采众长。

## 6.2 我们为什么需要以词为中心的搜索

如前所述，用户在搜索时通常并不关心文档是如何被分解成各个字段的。许多搜索用户都期望将文档看成一个单一的整体：匹配的搜索词越多，文档的相关性就应该越高。也许大家会觉得有点吃惊，搜索引擎中实现这一思路的功能出现较晚。相反，基于 Lucene 的多字段搜索（multifield search）使用的却是以字段为中心的技术。这种搜索技术将字段（而非搜索词）的评价值作为排名函数的核心。本节我们将详细探究为什么以字段为中心的方法会有相关性问题的。大家将会看到，以字段为中心的排名函数会导致出现两个问题：

- “白化象（*albino elephant*）”问题——有更多搜索词匹配的文档没有被排在更靠前的位置。
- 信号冲突（*signal discordance*）——对相关性的评价依赖于文档中那些不太直观的部分（例如，选择标题，而非正文进行评价），而不考虑整篇文档，或者大多数更为直观的部分（例如，整篇文章的文本，或影片的相关人员）。

接下来，我们将探讨上述这些问题。通过本节的学习，我们将能够准确地识别出，什么时候以字段为中心的搜索将无法我们的需求，而以词为中心的方法可能会取而代之。

### 6.2.1 猎寻“白化象”

如果我们搜索“Paul McCartney Concert near San Francisco”，但是却搜到了一大堆坐落在 San Francisco 的售卖 Paul McCartney 音乐的商店，大家有何感想呢？更糟糕的是，如果我们要翻到第二页才能找到期望的搜索结果，那该怎么办：那是一场在 San Francisco 附近举办的 Paul McCartney 音乐会！无论出于什么原因，搜索引擎

忽略了我们搜索中的主体部分——音乐会本身，却只把满足部分查询条件的结果排在了前面：

Search: Paul McCartney Concert Near San Francisco

Results:

1. CDs R' Us, San Francisco - Paul McCartney
2. MP3s By The Street Corner, San Francisco - Paul McCartney
- ...

Page 2:

- 19: Stubbys Music Emporium, San Francisco - Paul McCartney
20. Concert at Great American Music Hall, San Francisco - Paul McCartney

如果我们的搜索应用中存在这样的行为，那将是何其灾难性的后果？用户提供搜索词给我们，而我们却似乎视而不见！非常不幸的是，以字段为中心的搜索确实存在这样的问题。2004年，工程师 Chuck Williams 在他使用 Lucene 以字段为中心的搜索工具时发现了这一问题。在以字段为中心进行搜索时，他发现只匹配部分搜索词的搜索结果，其排名竟然会胜过匹配全部查询词的搜索结果。这一问题，也就是我们所说的“白化象”问题，对我们的搜索解决方案造成了严重的破坏。它是许多相关性问题的根源。当老板走到跟前，抱怨我们的方案似乎漏掉了某些用户想要的搜索结果时，那就有可能是“白化象”问题引起的，而以词为中心的搜索则可能是针对这一问题的解决方案。

那么，这个奇怪的名字是从何而来的呢？白化象一词源于 Chuck 用来阐述这一问题的一个典型例子。请看清单 6.1 中列出的文档，它们在 Elasticsearch 中建了索引。

#### 清单 6.1 对“albino elephant”的文档进行索引

```
PUT albinoelephant/docs/1
{ "title": "albino", "body": "elephant" }

PUT albinoelephant/docs/2
{ "title": "elephant", "body": "elephant" }
```

标题是“albino”且正文是“elephant”的文档

标题和正文都是“elephant”

使用以字段为中心的 `most_fields` 搜索技术，针对标题 (`title`) 和正文 (`body`) 搜索“albino elephant”，上面两篇文档中的哪一篇会排在前面呢？也许大家认为文档 1（标题 albino，正文 elephant）会比仅提到 elephant 的另一篇文档排名更加靠前。然而，执行搜索以后，其结果却截然不同，如清单 6.2 所示：

## 清单 6.2 恼人的 albino elephant 搜索

```
GET albinoelephant/docs/_search?
  "multi_match": {
    "query": "albino elephant",
    "type": "most_fields",
    "fields": ["title", "body"]}}}
```

以字段为中心的“albino elephant”搜索

Score	Title	Body
0.06365098	elephant	elephant
0.06365098	albino	elephant

汇总结果

“普通象”和“白化象”的结果相同

啊，“albino elephant”竟然没有任何额外的加分！为什么会这样呢？以字段为中心的搜索，并没有将匹配其中一个字段的 albino 与匹配另一个字段的 elephant 结合起来。most\_fields 在进行评价时没有考虑到，某个搜索词出现在一个字段里，而另一个搜索词出现在另一个字段里的情况，如图 6.3 所示。

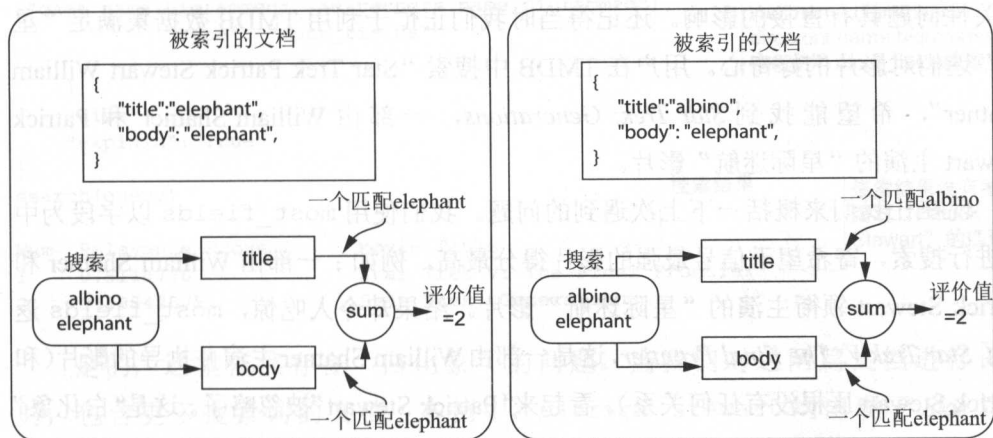


图 6.3 以字段为中心的搜索忽略了多个搜索词发生匹配的情形

每个字段的搜索都是彼此孤立的。要记住，以字段为中心的搜索在合并搜索结果之前，会将整个搜索字符串分发给各个字段进行评价。对标题的搜索（title:albino title:elephant）和对正文的搜索（body:albino body:elephant）是独立计算的，这两个字段的搜索，彼此之间并没有交互。elephant 同时匹配两个字段，还是 albino 匹配其中一个字段，elephant 匹配另一个字段，这二者之间并没有什么区别。我们可以将针对两篇文档的排名计算重新表述如下（匹配的搜索短语以粗体标明）：

```
(title:albino title:elephant) +  
(body:albino body:elephant) == score for two matches
```

```
(title:albino title:elephant) +  
(body:albino body:elephant) == score for two matches
```

换句话说，对于多个搜索词在这两个字段上同时出现的情况，以字段为中心的搜索并没有过多被“关照”。

理解“白化象”现象是尊重用户所有搜索词的基础。无视用户的搜索词可能会让我们为用户提供的搜索结果显得愚蠢至极。随着我们深入探究以词为中心的方法，大家将会看到，此类方法是如何解决“白化象”问题的，从而帮助我们免于被愤怒的用户蜂拥而至，上门指责！

## 6.2.2 在“星际迷航”的例子中寻找白化象问题

说到大批愤怒的用户，这表明了“白化象”现象对于我们在上一章遗留下来的相关性问题的影响。还记得当时我们正忙于利用 TMDb 数据集满足“星际”迷们对影片的好奇心。用户在 TMDb 中搜索“Star Trek Patrick Stewart William Shatner”，希望能找到 *Star Trek: Generations*，一部由 William Shatner 和 Patrick Stewart 主演的“星际迷航”影片。

现在让我们来概括一下上次遇到的问题。我们使用 `most_fields` 以字段为中心进行搜索，寄希望于信号最强的影片得分最高，例如：一部由 William Shatner 和 Patrick Stewart 领衔主演的“星际迷航”影片。结果却令人吃惊，`most_fields` 返回了 *Star Trek V: The Final Frontier*，这是一部由 William Shatner 主演并执导的影片（和 Patrick Stewart 压根没有任何关系）。看起来“Patrick Stewart”被忽略了。这是“白化象”问题导致的吗？让我们重新来回顾一下以字段为中心的技术细节，很快就会发现其中的问题！

大家可能还记得，在第5章中我们提到过，`most_fields` 搜索是考虑多个信号的一种理想的搜索方式。结合 `most_fields`，我们已经实现了一系列智能的信号和特征建模。利用如下这些字段，我们可以更加准确地对演职人员的名字做出评价：

- `cast.name.bigrammed`——其得分表明演员匹配与否。
- `directors.name.bigrammed`——其得分表明导演匹配与否。

大家回忆一下，`*.bigrammed` 字段存储的是双联词（两个单词构成的词组），



而非一个个独立的单词。以此作为 token，有助于我们建立更为准确的信号，用以查找与 patrick stewart 匹配的人，而非与单个词 partick 和 stewart 匹配的人。我们同时也会对影片的 title 和 overview 字段进行搜索，以支持用户根据片名和描述查找影片。

记住这一点之后，当我们再次利用第 5 章介绍的 most\_fields 进行搜索时，会发生什么情况呢？为什么它有可能不像用户期望的那样工作呢？这里隐藏着“白化象”问题吗？参见清单 6.3。

### 清单 6.3 Kirk 和 Picard<sup>1</sup> 到访了 albino elephants 星球

```
usersSearch = "star trek patrick stewart william shatner"
query = {
  "query": {
    "multi_match": {
      "query": usersSearch,
      "fields": ["title", "overview",
        "cast.name.bigrammed", "directors.name.bigrammed"],
      "type": "most_fields"
    }
  },
  "size": 5,
  "explain": True
}
search(query)
```

用户的查询

针对title、overview、cast.name.bigrammed、directors.name.bigrammed 字段进行most\_fields搜索

Num	Relevance Score	Movie Title
1	0.5114776	Star Trek V: The Final Frontier
2	0.38542575	Star Trek: Generations

搜索结果

搜索结果没有考虑对“Patrick Stewart”的匹配

是的，这里依然存在“白化象”的问题。当我们对这两篇文档进行评价时，包含更多搜索词的文档（本例中，patrick stewart 均出现于 *Star Trek: Generations*）在评价时并不具备任何优势。对前述清单中的评价进行分解（重点在三个发生匹配的字段上），我们得到的评价结果来自于如下以粗体标识的匹配：

```
(title:star title:trek title:william title:shatner...) +
(... directors.name.bigrammed:william shatner ...) +
(... cast.name.bigrammed:william shatner ...)
== total score for four matches

(title:star title:trek title:william title:shatner...) +
(... directors.name.bigrammed...) +
(... cast.name.bigrammed:patrick stewart cast.name.bigrammed:william shatner
...) == total score for four matches
```

<sup>1</sup> Kirk和Picard分别是星际迷航的两代舰长。——译者注

正如前面那个 *albino elephant* 的例子那样, *Star Trek: Generations* 考虑了用户指定的所有查询条件 (即 *patrick stewart* 在其中出现的情况), 但它和 *Star Trek V* 在排名上却没有任何差别。二者对总评价的计算, 依据的都是每个字段上的匹配, 而匹配全部搜索词的文档在排名分数上并没有什么优势。我们又遇到“白化象”问题了! 在尝试以词为中心的搜索方法时, 我们将再次使用这个“星际迷航”的例子, 用它来评估以词为中心的搜索是否真正解决了我们的问题。不过在此之前我们还得讨论一下有关以字段为中心进行搜索的另一个大问题。

### 6.2.3 避免信号冲突

如果我们搜索过学术论文的目录, 大家会关心匹配是否出现在论文的哪些特定的部分吗? 例如, 我们会优先考虑论文引言部分的匹配, 然后才是结论部分的匹配吗? 如果换一种划分方式, 将文档按页 (第一页、第二页, 以此类推) 拆分到各个可搜索的字段中, 那么情况又会如何呢? 我们如何为每一页按优先级排序呢? 关键在于, 我们的数据库可以用各种不同的方式对文章的文本进行随意划分! 但这些划分对搜索适用吗? 它们与用户对内容的“心智模型 (mental models)”相匹配吗? 实际上, 搜索用户对内容的期望, 可能远比我们给搜索引擎输入的那些源自系统的细粒度划分要宽泛得多。

构建与用户对内容的理解相一致的信号, 是信号建模的一项主要工作, 也是以词为中心进行搜索的一个主要特征。用户不会拘泥于数据库、解析器, 或 API 的细节。正如我们在前一章中学到的那样, 源数据模型里不存在为搜索而建的字段。然而, 在前面有关论文的例子中, 还存在着一个更加突出的问题: 信号冲突 (signal discordance), 如图 6.4 所示。信号冲突是指, 由搜索引擎中细粒度的、特定字段所产生的信号 (取自源数据模型) 与用户对内容的、更为宽泛的“心智模型”, 二者之间出现的不一致。因此, 信号冲突是信号建模的一种专门的缺陷, 即: 用户希望搜索引擎以较为宽泛的方式来工作, 而搜索引擎所产生的信号却无法对此进行度量, 因为我们仍然局限于数据库、API、解析器或其他源数据模型的具体细节当中。

通过对搜索词逐一进行处理的方式, 以词为中心的搜索方式可以帮助我们解决信号冲突的问题。相反, 以字段为中心的搜索方式更容易放大信号的冲突。当我们构造的字段过于精确地反映源数据模型时, 以字段为中心的搜索结果就无法与用户的期望相匹配。我们将会看到, 以词为中心的搜索, 通过对横跨多个字段的搜索评

价加以泛化，有助于我们更接近于用户对文档结构的朴素理解，以此对相关性的度量。

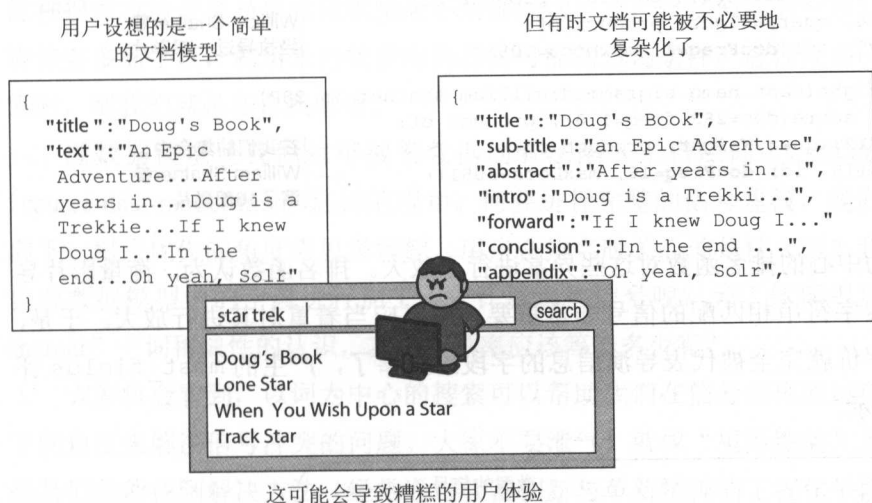


图 6.4 信号冲突：没有根据用户对内容结构的理解进行排名，由于以字段为中心的搜索关注的是不太常见的查询条件，因此这样做就可能产生一系列奇怪的结果。

### 6.2.4 理解信号冲突的机理

让我们花点时间来准确理解一下，以字段为中心的排名函数中存在的信号冲突问题。只有理解了这一点，我们才能对以词为中心的搜索方法能够在多大程度上有效地帮助我们做出评估。

信号冲突的问题，表现为对字段的评价无法满足用户的一般性预期。由于字段对排名的统计是由其自身来维护的，比如词的文档频率，因此，那些无法满足用户一般性预期的字段会严重破坏排名的结果。

针对“Star Trek Patrick Stewart William Shatner”这条失败的搜索，如果检查下面的解释信息我们就会发现，导致由 William Shatner 执导的影片被直接排在最前面的另一个主要原因是：虽然他只执导过一部影片，但是却参演无数。william shatner 在 `directors.name.bigrammed` 中的文档频率恰好为 1。该搜索词在这一字段中的匹配会引起我们对此字段的  $TF \times IDF$  评价出现异常，从而导致他所执导的影片被直接排到了最前面。

从清单 6.4 中我们可以看到，信号冲突是如何导致评价异常的。

## 清单 6.4 涉及 directors.name:william shatner 与 cast.name 的解释信息片段

```

1.8605413, weight(directors.name.bigrammed:william shatner in 282)
1.8605413, score(doc=282,freq=1.0), product of:
  0.22335224, queryWeight, product of:
    8.330077, idf(docFreq=1, maxDocs=3051)
0.2568409, weight(cast.name.bigrammed:william shatner in 282)
0.2568409, score(doc=282,freq=1.0), product of:
  0.1659712, queryWeight, product of:
    6.1900115, idf(docFreq=16, maxDocs=3051)

```

William Shatner 曾经执导过一部影片

在我们的集合中，William Shatner 参演了16部影片

以字段为中心的排名函数对这些异常进行了放大。排名函数认为，衡量影片导演是否与搜索字符串相匹配的信号非常重要，所以应当着重对其进行放大。于是，对该文档的评价就完全被代表导演信息的字段给主宰了，产生的 `most_fields` 评价如图 6.5 所示。

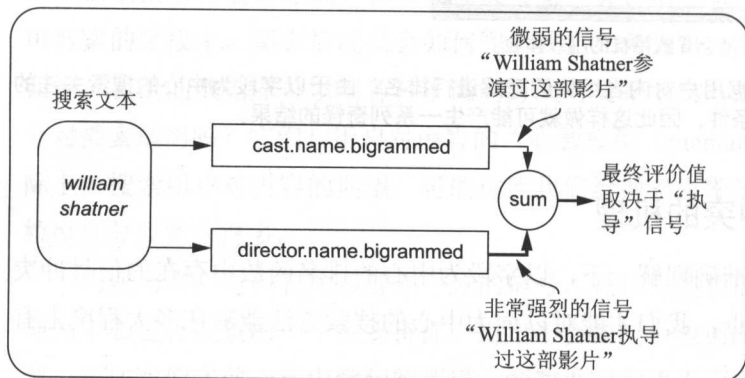


图 6.5 以字段为中心的方法对源数据模型过于细粒度的字段划分导致其评价与用户期望不相符

可是，用户真的关心这些吗？针对 `directors.name.bigrammed` 的评价，一个最终对整体评价起决定作用的字段评价，真实地反映了用户对 William Shatner 是否执导过某部影片的看法吗？答案几乎是否定的。大部分“星际”迷们可能都不太关心 William Shatner 到底是执导还是参演了一部影片这两者之间的区别。而且，即使“星际”迷们关心这一点，他们可能也更加关心由其主演的影片。

当字段直接来自源数据模型时，由搜索得到的  $TF \times IDF$  计算结果并不能保证恰好能满足用户的期望。如果字段（以及未经加工处理的部分底层特征）在排名时没有与用户所关心的信号相符，那么  $TF \times IDF$  就有可能是一种非常糟糕的度量指标。如果我们只是因为“数据库长得就是这个样子”，就对这么多个字段进行搜索，

那最终我们就会得到许多类似于这样的，无法与用户期望相符的异常评价。

以字段为中心的搜索放大了信号冲突。由于每个字段的评价都是彼此孤立的，这种搜索方法很容易导致搜索结果不是偏向这一方，就是偏向另一方。当我们面对许许多多的字段，利用来自数据库的各种可能的查询条件，进行以字段为中心的搜索时，那我们就是在自找麻烦。

源数据模型使我们对可搜索数据的看法陷入了自底向上、源数据模型优先（source-data-model-first）这样的视角。而要进行完整的信号建模，我们需要从自顶向下、用户优先的角度来思考问题。用户在排名时关心的是什么呢？我们如何对取自源数据模型中的字段进行加工，来计算这些信号呢？为了体现用户对“william shatner”一词稀缺性的认识，其文档频率应该等于多少呢？

大家将会看到，以词为中心的搜索可以帮助我们在信号建模的过程中从自顶向下的角度来解决信号冲突的问题。大家不要泄气！就像“星际迷航”里，企业号的船员们总能找到解决方法（经常通过极富创新与革新精神的工程化手段）一样。我们也能！让我们扬帆起航，去探索以词为中心的搜索这一陌生的全新领域吧！

## 6.3 完成第一个以词为中心的搜索

我们已经看到了，以字段为中心的搜索会导致各种各样的问题。现在，我们将有幸体验以词为中心的搜索带给我们的全新世界，如图 6.6 所示。

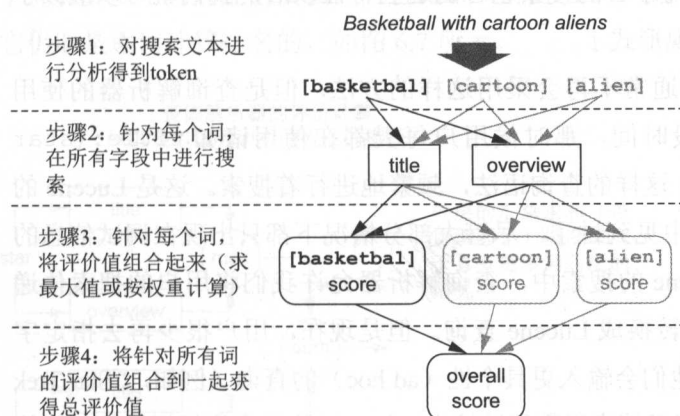


图 6.6 以词为中心的搜索将搜索词逐一放入每个字段，以此得到针对该词的相应评价。

以词为中心的搜索从自顶向下的角度出发看待搜索，解决了“白化象”问题和

信号冲突：它将搜索词拆分开来，然后在—组字段中逐一查询每一个词。以词为中心的搜索提示我们要从用户的角度出发，多加关注与用户查询词相关的信号，而不仅仅只关注字段，我们会发现，这样做可以简化信号建模的诸多方面。以这样的方式产生的信号，可以解决更为广泛的、自顶向下的问题，更接近用户对内容的通俗理解，而不是对源数据模型在结构方面的理解。以词为中心的搜索解决了许多其设计之初想要解决的问题，但是它本身也存在一定的问题。在本节接近尾声的时候，大家会逐渐开始意识到，我们需要的是一种组合的方法。

### 6.3.1 使用以词为中心的排名函数

在讲解以词为中心的搜索方法之前，我们得先来看一下，在解决以字段为中心的搜索问题时，人们所采用的诸多原始方法中的其中一种：以词为中心的查询解析器（term-centric query parser）。这种查询解析器会对查询字符串进行解析，并在搜索各个字段之前，将查询词提取出来，通过这样的方式来实现以词为中心的搜索。这一方法对 Solr 用户来说非常熟悉（Solr 完全依赖于查询解析器），而 Elasticsearch 在 `query_string` 查询中也使用了查询解析器。

通过探索查询解析器的工作原理，我们将大致了解以词为中心的搜索是如何工作的。以词为中心的方法究竟有哪些常见的基本特征？以词为中心的搜索，其强大的威力源自何处？它如何让搜索结果偏向含有更多搜索词的文档？得到这些好处的代价是什么？当了解了以词为中心的搜索包含的这些特性以后，我们就可以继续学习它的一些更为现代化的表现形式了。

尽管现在的搜索应用中通常不再会采用这样的方法，但是查询解析器的使用可以追溯到过去很长的一段时间，那时候用户每天都在使用诸如 `title:(star trek)+overview:shatner` 这样的查询语法，频繁地进行着搜索。这是 Lucene 的查询语法，我们已经在本书中见到过了，尽管大部分情况下都只出现在调试信息的上下文中。在早期基于 Lucene 的搜索中，查询解析器允许我们将用户的搜索传递给搜索引擎，搜索引擎将其转换成 Lucene 查询。但是现在，用户很少再去指定字段或布尔操作符了。相反，他们会输入更具个性（ad hoc）的查询（例如，“Star Trek Patrick Stewart”），然后寄希望于搜索引擎能够搞定一切。以往的查询解析器就已经能够做到，将这些特别的搜索转换成逐词的布尔查询（term-by-term Boolean query）。

查询解析器具有以词为中心的优势，主要原因在于它侧重于针对搜索词的一个



个布尔查询。如果我们只有一个字段要搜（例如 overview 字段），那么用户期望的行为相当于将“Star Trek Patrick Stewart”变成这样一条更为精确的查询 `overview:star overview:trek overview:patrick overview:stewart`（一条具有 4 个 SHOULD 子句的查询）。用户通常认为匹配的词越多，得到的评价就应该越高——这恰好就是我们使用布尔查询所能得到的结果。基于布尔查询的评价，会非常偏向于匹配更多搜索词的文档。还记得吗？布尔查询会将底层评价相加求和，然后乘以一个协调因子，该协调因子以乘数的方式，大大地削弱了那些不满足所有查询子句的文档，因而会进一步降低不能匹配全部查询词的那些文档的排名。所以，布尔搜索中的每个布尔子句都对应于一个搜索词，这样的查询表现出强大的以词为中心的行为，搜索词匹配越多的文档，其排名越靠前，而搜索词匹配越少的文档，其权重则会下降。

在前面的例子里，只有 overview 一个字段是可搜索的。如果引入更多的字段，会发生什么情况呢？查询解析器如何才能实现以词为中心的搜索，不论出现几个字段都能应对自如呢？答案是 `dismax` 查询（`dismax query`）。

`DisjunctionMaximumQuery`（简称 `dismax`）为我们在第 5 章中介绍过的基于 `best_fields` 的多字段搜索策略提供支持。在 Lucene 的查询语法中，到目前为止我们已经多次看到过，符号“|”的意思是指“取最高评价”。`best_fields` 利用 `dismax` 方法来获取字段的最高评价，而 `query-parser` 查询则在逐词查询的基础上利用 `dismax` 来选择针对每个查询词得分最高的字段。类似的，布尔查询更加偏重于拥有更多子句匹配的文档，但它仍旧和前面的单字段例子道理一样。从最外层看，它仍旧是逐词计算排名的，如图 6.7 所示。

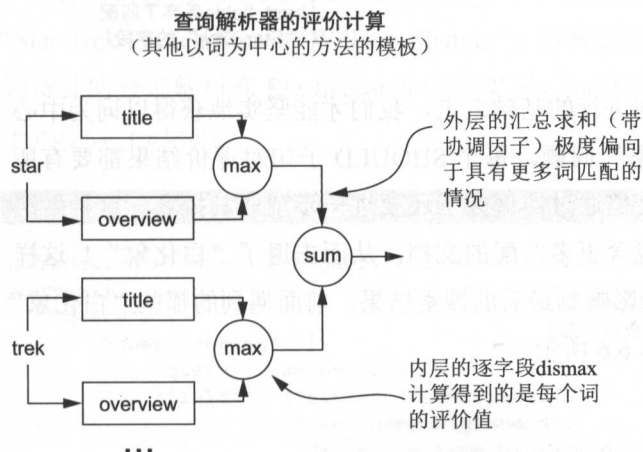


图 6.7 查询解析器实现了以词为中心的搜索：内部采用逐字段的 `dismax` 计算，在评价计算上则偏重于拥有更多词匹配的以逐词方式进行的布尔查询。

相比于 `best_fields` 搜索，在最内层对每个字段进行布尔查询计算。此处的放大处理 (`boost`) 只针对拥有更多词匹配的单个字段，而非整篇文档。

这里列出的是在计算 `dismax` 之前，执行两条布尔查询的 `best_fields` 搜索：

```
(overview:star overview:trek overview:patrick overview:stewart) |
(title:star title:trek title:patrick title:stewart)
```

利用查询解析器，我们获得了逐词进行的 `dismax` 搜索，它是在逐词而非逐字段的基础上进行布尔查询的：

```
(overview:star | title:star) (overview:trek | title:trek)
(overview:patrick | title:patrick) (overview:stewart | title:stewart)
```

翻译成排名函数，如下所示 (用  $S_{term}$  代表针对某个词的 `dismax` 操作结果)：

```
coord × (Sstar + Strek + Spatrick + Sstewart)
```

不过，`best_fields` 方法也存在“白化象”问题，如清单 6.5 所示。我们可能会遇到 `star trek` 匹配一个字段，而 `patrick stewart` 匹配另一个字段的情况。然而，因为 `best_fields` 只返回最优的字段评价，其中一个字段的匹配就会被忽略。这就可能会导致匹配更多搜索词的文档在排名时被忽略的情况。

#### 清单 6.5 `best_fields` 存在“白化象”问题

```
(overview:star overview:trek overview:patrick overview:stewart) |
(title:star title:trek title:patrick title:patrick)
```

best\_fields选择overview字段  
的评价值作为评价结果

best\_fields丢弃了匹配  
“Star Trek”的字段

只有让布尔查询立足于逐词进行的基础之上，我们才能坚实地获得以词为中心的搜索行为，从而避免“白化象”问题。每个 `SHOULD` 子句对评价结果都要有所贡献。而且，不匹配任何词的文档通过协调因子还要进一步削弱其排名。这种做法不管字段的匹配如何，而只看包含更多匹配的文档，从而击退了“白化象”！这样一来，我们可以保证每个词都能影响到最后的搜索结果，前面遇到的那些“白化象”现象就可以得到避免了，如清单 6.6 所示。

## 清单 6.6 逐词进行 dismax 计算使每个词都能对搜索结果产生影响

```
(overview:star | title:star) (overview:trek | title:trek)
(overview:patrick | title:patrick) (overview:stewart | title:stewart)
```

单词Star和Trek都对  
搜索结果产生了影响

针对Patrick和Stewart  
的词匹配也是如此

内部逐字段进行 dismax 计算，加上外部针对所有词的布尔查询，这样一种模式在以词为中心的搜索中是处于核心地位的。它不因为某个字段的强弱而以偏概全，而是在逐词进行的基础上构建信号。在后面的几节里，我们将看到这种模式给以词为中心的搜索带来的优劣得失。那么，我们可以就此止步了吗？查询解析器是整个拼图的全部吗？这样做能否满足“星际”迷的需求吗？或者说，这样做还不够？为了解答这些问题，让我们一起来进行查询试验吧。

## 6.3.2 运行以词为中心的查询解析器（深入底层）

前面我们了解了查询解析器的排名函数是如何工作的，这里让我们一起来看看，它是否能包治百病。让我们回到“星际迷航”TMDB的那个例子。或许查询解析器能满足“星际”迷们的需求？让我们从以字段为中心的 multi\_match 查询换到以词为中心的查询解析器，将那些有问题的查询重新执行一遍，结果会怎样呢？很快，我们就会惊奇地发现——不仅查询解析器，包括以词为中心的搜索在内，对它们的使用通常都会受到极大的限制。

让我们使用 Elasticsearch 的 query\_string 查询，来测试一下我们的问题查询“Star Trek Patrick Stewart William Shatner”。这种查询方法是具备以词为中心这一行为特征的查询解析在 Elasticsearch 中的实现。如清单 6.7 所示，该查询的接口类似于 multi\_match。

## 清单 6.7 运行 query-parser 查询

```
usersSearch = "star trek patrick stewart william shatner"
```

```
query = {
  "query": {
    "query_string": {
      "query": usersSearch,
      "fields": ["title", "overview",
        "cast.name.bigrammed",
        "directors.name.bigrammed"],
```

使用 query\_string 查询解析器进行逐词 dismax 计算

用户的查询

```

    },
    "size": 5,
    "explain": True
}
search(query)

```

#### Results

Num	Relevance Score	Movie Title
1	1.5344285	Star Trek IV: The Voyage Home
2	1.0545591	Star Trek: Nemesis
3	0.7853979	Star Trek
4	0.6283183	Star Trek: The Motion Picture
5	0.6283183	Star Trek: Insurrection

看到这样的结果，我们的“星际”迷们一定会抱怨，简直差远了！记住，我们的用户也许是想寻找 *Star Trek: Generations*，一部由 Patrick Stewart 和 William Shatner 主演的电影。问题出在哪儿了呢？检查一下查询验证服务，没错啊，这确实是以词为中心的搜索。但是，令人吃惊的是，我们希望看到的字段竟然会被漏掉：

```

(title:star | overview:star) (title:trek | overview:trek)
(title:patrick | overview:patrick) (title:stewart | overview:stewart)
(title:william | overview:william) (title:shatner | overview:shatner)

```

我们明确地告诉 query\_string 查询，去搜索双联词字段 cast.name 以及 directors.name。然而，看上去似乎系统并没有搜索这两个字段。这是为什么呢？问题出在哪儿了呢？

### 6.3.3 理解字段同步

如果我们仔细想一想，这怎么可能呢？大家是否还记得，对于字段 \*.bigrammed 而言，搜索词所采用的字符与那些文本字段是完全不同的。对于双联词字段，搜索词以两个单词构成的词组“william shatner”为单位，而普通的文本字段则以“william”或“shatner”为单位。由此，查询解析器是怎么知道该如何对其进行处理的呢？我们能在以词为中心的搜索方案中继续保持 \*.bigrammed 字段的双联词特性吗？

让我们深入思考一下这个问题。一个简单的办法是，或许我们可以让查询解析器尝试搜索每一个独立的文本词（text term），例如：

```
... (title:william | overview:william | directors.name.bigrammed:william)
```

但是这有些讲不通。此处，william 肯定不是双联词字段的匹配项。我们显式

地构造双联词字段，就是为了让系统能够利用双联词对其进行搜索，从而给搜索以特定的信号。然而，因为我们没有选择双联词，从而限制了对这一字段的使用，以词为中心的搜索采取这样的手段，使我们当初构造该字段的价值大打折扣！

要是我们能够找到方法，可以按期望的搜索词对双联词字段进行搜索，情况会怎样呢？假如用下面的方法，情况又会如何呢？

```
... (title:william | overview:william |  
    directors.name.bigrammed:william shatner) ....
```

但这不是一个同类型的一一对应的搜索词比较（apples-to-apples search-term comparison）！这也不是以词为中心的搜索！以词为中心的搜索，其基本原则是将每个像 william 这样的搜索词分离开来。此处，我们将搜索词 william 与 shatner 合并到了一起，用于针对 directors.name 的双联词查询。这样一来，shatner 一词就会影响到针对 william 的搜索评价了。

因此，在实际使用中，以词为中心的查询解析器强制每个字段使用相同类型的搜索词。我们将搜索的这一属性称之为字段的同步性（field synchronicity），即以同类型的搜索词来进行多字段查询的能力——换言之，也就是限制所有字段都必须以同样的方式进行搜索。在本例中，Elasticsearch 剔除了那些无法使用通用查询解析器的字段，选择去掉了双联词字段。通过这种方式，保证对所有字段的搜索都采取了统一的、以词为中心的方式。这是一种常用的，或许也是最为安全的，解决这一类问题的方法（尽管这样做可能会因为缺少有帮助的错误信息而为人所诟病）。

### 6.3.4 字段同步和信号建模

对字段同步性的要求意味着什么呢？在前面的章节中，我们强调利用分析手段来产生具有排名信号的字段，以供人们使用。我们努力地构造信号模型，使搜索引擎能够对重要的用户排名规则进行度量。我们说过，每个字段可能都是一片独特的雪花，在搜索过程中以其独有的方式，为排名函数提供独特的信号。在第 4 章中，我们已经见到过好几个这样的例子了，我们向大家展示了很多分析技巧，可以产生出能够衡量用户意图的独特字段。而以词为中心的搜索强制每个字段在行为上保持一致，这样做是否会妨碍我们对信号进行建模呢？

是的，这可能会是一个问题。我们知道，在解决以字段为中心进行搜索时遇到的那些问题，在以词为中心的搜索中是必不可少的。但是，我们也需要这些字段能

够自身工作良好才行。否则，我们拿什么匹配地理位置呢？还有人的名字？观点？分类标准，或者作为相关性技术工程师需要处理的任何其他类型的搜索实体呢？难道搜索引擎就这样陷入了两个不甚明智的极端了吗？一边是允许我们以普通的、逐词的方式进行搜索，为了建立信号，可以扼杀某个字段的影响力，另一边则是以字段为中心，存在烦人的“白化象”和信号冲突问题。

是否有方法能够鱼与熊掌兼得呢？这是非零即一的选择吗？还是可以从 A 中取一部分，再从 B 中取一部分——将两者的优势结合在一起，形成满足我们需求的解决方案呢？

这是以词为中心的搜索与以字段为中心的搜索，两者之间在基本层面上的一种“阴阳互补”，有利有弊。现在让我们把这个问题放在一边。暂且将这一冲突当作是“最后的边疆”吧。正如“星际迷航”里那样，我们将寻找方法，把彼此对立的两种思想放到一起，共同构建出更为和谐、更加有用的东西！

### 6.3.5 查询解析器和信号冲突

由于忽略了字段的同步性，我们之前所做的工作似乎是一个不太成功的开端。让我们再来试一试，看看以词为中心的搜索是如何工作的。也许，直接对非双联词的、常规的文本名称字段进行匹配也不是什么糟糕的事情，如清单 6.8 所示。

清单 6.8 保持字段同步性的搜索

```
usersSearch = "star trek patrick stewart william shatner"
query = {
  "query": {
    "query_string": {
      "query": usersSearch,
      "fields": ["title", "overview",
                "cast.name", "directors.name"],
    }
  }
}
search(query)
```

← 用户的查询

← 搜索非双  
联词字段

Results:

Num	Relevance Score	Movie Title
1	3.03831	Star Trek V: The Final Frontier
2	2.282416	Star Trek: Generations
3	1.7969469	Star Trek: Nemesis
4	1.4064612	Star Trek IV: The Voyage Home
5	1.3728689	Star Trek: The Motion Picture



这样做让结果有了一定的改善，但仍然没有达到预期的效果。我们的目标结果，*Star Trek: Generations* 排在了第二位，但令人吃惊的是，没有 Patrick Stewart 参与的电影却排在了第一位。这又是由“白化象”问题引起的吗？我们为那些包含更多匹配词的文档赋予更高的权重了吗？通过查看查询验证服务以及解释信息，我们发现确实有额外的权重被分配给了包含更多匹配词的文档。我们正在解决“白化象”的问题。查询验证服务通过列出我们所期望的，以逐词方式进行的布尔查询，反映了这一情况：

```
(title:star | overview:star | cast.name:star | directors.name:star)
(title:trek | overview:trek | cast.name:trek | directors.name:trek)
(title:patrick | overview:patrick |
    cast.name:patrick | directors.name:patrick) ...
```

针对我们所期望的搜索结果，*Star Trek: Generations*，其汇总之后的解释信息给出了如下所示的计算过程：

```
2.282416, sum of:
  0.78420293, max of:
    0.78420293, weight(title:star in 847)
    0.12733683, weight(overview:star in 847)
  0.9435517, max of:
    0.9435517, weight(title:trek in 847)
  0.08256196, max of:
    0.08256196, weight(cast.name:patrick in 847)
  0.14055088, max of:
    0.14055088, weight(cast.name:stewart in 847)
  0.060675804, max of:
    0.060675804, weight(cast.name:william in 847)
  0.2708729, max of:
    0.2708729, weight(cast.name:shatner in 847)
```

不过，尽管有较多数量的匹配项，可在排名计算中，该文档仍然没有胜出。这是什么原因导致的呢？而且当我们查看 *Star Trek V: The Final Frontier* 时，还注意到有一个协调因子，因为没有匹配所有搜索词而对其进行了惩罚。这一定另有隐情：

```
3.03831, product of:
  4.557465, sum of:
    0.68617755, max of:
      0.68617755, weight(title:star in 210)
    0.8256078, max of:
```

因为6个词中仅有4个匹配，所以协调因子对其进行惩罚

```

0.8256078, weight(title:trek in 210)
1.0834916, max of:
0.07281096, weight(cast.name:william in 210)
1.0834916, weight(directors.name:william in 210) ← “william” 和 “shatner” 针对导演的匹配，其评价价值格外高
1.9621884, max of:
0.32504746, weight(cast.name:shatner in 210)
1.9621884, weight(directors.name:shatner in 210) ←
0.6666667, coord(4/6)

```

结果表明，此处主要的差别在于 william shatner 在 directors.name 字段中的匹配。还记得我们讨论的信号冲突吗？问题的根源在于：用户可能并不关心 William Shatner 是否是导演。然而因为他只执导过一部影片，导致其针对于导演的匹配，对应的  $TF \times IDF$  评价价值相对其他匹配高出了很多。尽管布尔查询针对其没有匹配所有查询词的情况进行了“严厉的判罚”，但是针对 directors.name 的  $TF \times IDF$  评价还是令总体的字段评价偏高了。

出现这种情况的原因在于，尽管查询解析器解决了“白化象”问题，但是它并没有解决信号冲突问题。查询解析器并没有考虑根据每个字段的文档频率，对搜索词的整体评价给予调整。它并没有做出直接的调整，以修正 directors.name 的文档频率，从而更为准确地反映出用户的真实期望。我们在排名函数中所使用的信号，与用户对这些搜索词的总体期望并不相符。人们陷入了源数据模型的细节之中。

### 6.3.6 对以词为中心的搜索进行调优

大家可能还记得在第5章中，在 best\_fields 搜索中，针对字段评价的不一致性问题，一种可行的解决方案是通过放大处理（boosting）来明确声明对某个字段的偏好。现在让我们花点时间来看一下调优的问题。以词为中心的搜索中有类似的工具和手段，可以让我们根据用户需求对排名函数做出修改吗？我们将会看到，由于众多以词为中心的方法之间具有相似性，所以对某种方法的调优，通常也可以被应用于另一种方法。基于这一点，我们如何才能使信号更加接近用户的搜索期望呢？或许我们可以简单地把信号冲突问题整个绕过去？

看看这个以词为中心的排名函数，似乎我们已经无法对任何参数施以更多的影响了：

```
coord × (Sstar + Strek + Spatrick + Sstewart + Swilliam + Sshatner)
```

但是，一定有某种办法可以对这个公式做出调整，以解决信号冲突的问题。办法当然有。我们知道，每个词的信号（例如  $S_{star}$ ）都是对所有被搜索字段计算

dismax 值得到的。回忆一下前一章中提到的 `best_fields` 的做法，排名函数允许我们通过 `tie_breaker` 和针对每个字段的放大处理对其进行调优。幸运的是，那些工具和方法在这里也是同样适用的。此处，`B` 代表 `boost`，`S` 代表 `score`：

$$S_{\text{star}} = B_{\text{title}} \times S_{\text{title}} + \text{tie\_breaker} \times (B_{\text{overview}} \times S_{\text{overview}} + \dots)$$

对于搜索词的评价计算，我们可以自行决定是否：

- 在 `dismax` 的计算公式中通过设置 `tie_breaker`（默认值为 0），将没有胜出的字段纳入评价。
- 通过 `boost` 值（即 `Boverview`），为每个字段所产生的影响定义权重（或重新寻找平衡）。

对以词为中心的搜索进行调优，主要是指在每一个词的评价结果中针对 `dismax` 计算的微调。我们在前一章中针对以字段为中心的搜索制订的调优策略在这里仍然是适用的。那么，我们还是基于搜索词的“最佳字段（best field）”逐个计算每个词的评价值吗（根据需要通过 `boost` 值强化某一方面的偏好）？或者倾向于使用 `most_fields` 方法进行 `dismax` 计算吗（因有多个字段与搜索词匹配而提高其评价值）？

在信号冲突的例子中，我们可以利用 `boost` 值强制让某个字段的评价值高于其他字段——正如我们在前一章中对 `best_fields` 采取的措施一样。在清单 6.9 中，我们将放大对 `cast.name` 的评价来抑制 `directors.name` 的评价，强制让演员的匹配优于导演的匹配。

#### 清单 6.9 对以词为中心的搜索进行调优

```
usersSearch = 'star trek patrick stewart william shatner'
query = {
  "query": {
    "query_string": {
      "query": usersSearch,
      "fields": ["title", "overview",
        "cast.name^10", "directors.name"],
    },
  },
}
search(query)
```

用户的查询

将cast.name放大10倍

Results:

Num	Relevance Score	Movie Title
1	1.0714334	Star Trek: Generations
2	0.8237567	Star Trek V: The Final Frontier

3	0.5298387	Star Trek II: The Wrath of Khan
4	0.52354753	Star Trek IV: The Voyage Home
5	0.502342	Star Trek: Nemesis

这正是我们想要的结果！但这是解决信号冲突的最佳方法吗？我们已经给了评价算法一个“下马威”，迫使其接受我们对字段的偏好。这种解决方法是长久之计吗？

尽管这是解决信号冲突的一种可行的方案，但它并不是一种令人满意的方案（而且有可能是一种脆弱的方案）。“放大处理（boosting）”的思路是强制提高某个字段的评价值以超过其他字段，而不是去正面解决文档频率无法真正度量搜索词稀缺性的问题。如果用户想搜索的是一位只当过一次演员的导演，情况会怎样呢？这种解决方法会对导演有过一次参演经历的情况给予加分，而对该名导演曾经执导过的影片给予减分。这完全不是用户所期望的。

当我们需要声明偏好的时候，这些调优技术仍然可以移植到其他形式的、以词为中心的搜索中去。就像我们在第5章中所讨论的那样，让搜索评价在多大程度上依赖于针对搜索词的最佳字段，还是依赖于 `most_fields` 的行为，两者之间的矛盾（此处通过 `tie_breaker` 反映出来）仍然存在。

由于这些信号冲突问题的存在，我们要从查询解析器的这趟“列车”上中途下车了。以词为中心的查询解析器是被设计用来解决“白化象”问题的。它们原本可以让我们走得更远，但是由于缺少解决信号冲突问题的恰当手段，接下来我们将着重研究另外两种以词为中心的搜索方法。

## 6.4 在以词为中心的搜索中解决信号冲突

尽管 `dismax` 类型的查询解析器展示了以词为中心的搜索技术具备的一系列基本行为，但是还有另外两种以词为中心的方法也经常被用到。这两种方法分别是自定义全字段（`custom all fields`）和跨字段（`cross_fields`），它们同时解决了“白化象”问题和信号冲突问题。“自定义全字段”的方法在索引阶段将所有其他字段都整合到一起组成了一个总的字段（`one all field`），这样做虽然不怎么灵活，但确实有效。与之对应的，`cross_fields` 搜索则在查询阶段，通过在搜索之前调整字段的文档频率，试图直接解决信号冲突的问题。正如我们所看到的那样，这样做带来了更大的灵活性，其代价是引入了误差和不确定性。

### 6.4.1 将字段合并成自定义全字段

自定义全字段在索引阶段解决了以词为中心的搜索带来的问题。这种方法直接将我们想要搜索的字段合并为一个单一的字段。毕竟，如果只有一个字段，那么采用 `multifield` 方法的以字段为中心的搜索是绝对不会有问题的！起名为全字段是因为我们可以将全部字段复制到一起组成一个单一的字段，这个字段就被称为“全字段(all field)”。在 Solr 和 Elasticsearch 的社区里，人们也称其为拷贝字段(copy field)。例如，我们利用搜索引擎在索引阶段能够将多个字段加在一起的能力，可将遇到问题的名称字段 `cast.name` 和 `directors.name` 合并成一个更为普通的 `people.name` 字段。

不过，这样做看起来有点奇怪；这种做法对我们有什么好处呢？这个嘛，试想一下我们那位问题演员 / 导演 William Shatner，他执导过一部影片，但是却参演无数。这一信号冲突使其针对 `directors.name` 的匹配，得分过高。这与用户对文档结构的总体预期是矛盾的。通过将 `directors.name` 和 `cast.name` 合并成一个更为宽泛的衍生字段 `people.name`，其信号就会变得更具一般性，或许这更贴近用户的搜索期望。这种一般性，会在评价价值上有所体现。通过搜索 `people.name`，不论 William Shatner 是导演还是演员，对此二者之一的特殊偏好顷刻间荡然无存。这种差别也烟消云散了！与之相关的搜索词，其文档频率被合并在了一起，并反映出一个更为一般性的概念，即：William Shatner 作为一个人，而不是作为一名导演或是演员，与某部电影的关联程度如表 6.1 所示。

表 6.1 短语 “william shatner” 在两个源字段，以及合并后的自定义全字段 `people` 中的相关文档频率

	william shatner 的文档频率
<code>directors.name</code>	11
<code>cast.name</code>	16
<code>people.name (custom all field)</code>	16

与任何其他字段一样，我们通过带 `SHOULD` 子句的布尔查询对该全字段进行搜索。要记住，这种方式仍然偏向于包含更多匹配的文档，因此不存在令人担忧的“白化象”问题。

```
people.name:william people.name:shatner
```

下面让我们来演示一下，如何将这样一个自定义的全字段添加到我们的文档中。

我们将会看到，通过消除演员和导演之间的差异，搜索结果会更加接近用户的预期。

要建立一个自定义的全字段，首先需要在映射信息（mapping）中定义一个新的字段。定义 people 字段的方法与定义其他字段是一样的。然后，针对每一个为 people 字段提供“素材”的字段，在其映射项（mapping entry）中，使用 copy\_to 选项将源字段的内容复制到目标字段。由于映射信息过于冗长，因此我们在这里每次只为大家展示一条映射。首先来看 people 字段的映射，它看起来和我们在前一章中见到过的、针对其他名称字段的映射一模一样。我们甚至保留了这个字段的双联词特征，如清单 6.10 所示。

清单 6.10 自定义全字段 people 的映射信息

```
mappingSettings = {
  "movie": {
    "properties": {
      "people": {
        "properties": {
          "name": {
            "type": "string",
            "analyzer": "english",
            "fields": {
              "bigrammed": {
                "type": "string",
                "analyzer": "english_bigrams"
              }
            }
          }
        }
      }
    }
  }
},
```

对目标字段people的定义  
和其他字段的一样

people期望接收“name”  
属性的源字段

people依然保留  
“bigrammed”属性

继续看其他的映射，我们将 cast.name 字段复制到了 people.name 字段。对导演字段也做了同样的操作。清单 6.11 将 copy\_to 加到 cast 的映射信息中。

清单 6.11 添加从 cast.name 到 people.name 的 copy\_to

```
"cast": {
  "properties": {
    "name": {
      "type": "string",
      "analyzer": "english",
      "copy_to": "people.name",
      "fields": {
        "bigrammed": {
          "type": "string",
          "analyzer": "english_bigrams"
        }
      }
    }
  }
},
```

将该字段加入  
people.name



我们需要对所有数据重新建立索引：

```
reindex(analysisSettings, mappingSettings, movieDict)
```

太好了！如果现在我们对新的字段进行搜索，将会发生什么情况呢？有了这个新的字段，我们的选择就会变得非常灵活。我们可以在任何需要对某个字段进行搜索的地方用到它。不过目前，我们只是想看一下这个字段的使用方法，当直接用这个新的 `people.name` 字段来搜索两位演员时，请大家注意，清单 6.12 中将会发生些什么。

清单 6.12 自定义全字段的简单使用

```
usersSearch = 'patrick stewart william shatner'
query = {
  "query": {
    "match": {
      "people.name": usersSearch,
    }
  }
}
search(query)
```

匹配查询（搜索单个  
字段，`people.name`）

用户的查询

Num	Relevance Score	Movie Title
1	1.3773818	Star Trek: Generations
2	0.6629994	Showtime
3	0.65602106	The Wild
4	0.5989805	Bill & Ted's Bogus Journey
5	0.58601415	Star Trek V: The Final Frontier

首先我们注意到，同时包含两名演员的影片，*Star Trek: Generations*，排在了搜索结果的第一位。这并不奇怪；因为搜索两名演员的布尔查询，对于包含更多搜索词的文档，总是会推高其在搜索结果中的排名。搜索只在单一字段上进行这一事实，保证了我们不会再次遇到“白化象”问题。

信号冲突的问题在本例中也得到了解决。对于 `people.name` 而言，匹配 `william shatner` 的文档频率，其在 *Star Trek V* 中的取值与在 *Star Trek: Generations* 中的是一样的。因此，这样得到的评价结果更加接近于用户的期望。William Shatner 执导过 *Star Trek V* 这一情况，由于 `people.name` 字段经过合并的文档频率，基本上就被忽略了。而且，我们注意到 *Star Trek: Generations*，这部满足了所有查询条件的影片，其所得到的评价是如此之高。这是因为，针对组合字段的布尔查询，对包含所有查询词的文档进行了放大。

对于这种方法的使用，我们应该要达到什么样的程度呢？是否每个字段都应该被复制到一个全字段中呢？Elasticsearch 充分运用了这一方法。默认情况下，每一个字段都会被复制到一个包含所有内容的 `_all` 字段中。对本例而言，搜索字段 `_all` 即可返回我们想要的结果。

清单 6.13 搜索 `_all`

```
usersSearch = "star trek patrick stewart william shatner"
query = {
  "query": {
    "match": {
      "_all": usersSearch,
    }
  }
}
search(query)
```

← 用户的查询

Num	Relevance Score	Movie Title
1	0.9441141	Star Trek: Generations
2	0.577018	Star Trek: Insurrection
3	0.577018	Star Trek: First Contact
4	0.56560814	Star Trek V: The Final Frontier
5	0.5166054	Star Trek: Nemesis

然而，当我们继续使用同样的方法，只搜索 `patrick stewart` 时，排在最前面的搜索结果却与 `Patrick Stewart` 完全无关：

1	0.4262765	Panic Room
2	0.33741575	Conspiracy Theory
3	0.33741575	The Wolverine
4	0.33741575	Vertigo
5	0.33741575	Star Trek: Insurrection

出现这种情况是因为，在 `panic room` 经合并而成的字段 `_all` 里，文本 `patrick` 和 `stewart` 都被提到了。针对 `_all` 的搜索，其所得到的信号是模糊不清的，而且它也没有被映射成用户认可的、有具体意义的信息。而对于我们悉心构建的 `people.name` 字段而言，其信号则可能更接近于用户所认为的、有意义的查询条件；凡是与电影有关的人员，它都做了相关性评价的度量。“吸收”了所有文本的自定义全字段（就像 `_all` 字段那样），容易变得过于宽泛，通常应该避免使用。

事实再一次证明了，以词为中心的搜索并不是什么灵丹妙药。用户可能确实非常关心匹配了什么。为了严格控制每个字段的匹配，我们经常需要依靠以字段为中心的搜索，以及第4章和第5章中介绍的一些经验技巧。在本章的后面，大家将会

看到，我们是如何做到鱼和熊掌兼得的。

## 6.4.2 利用 cross\_fields 解决信号冲突

自定义全字段是静态的，是我们在对文档进行索引时建立的。与之相反，cross\_fields 搜索是动态的，它在查询阶段解决信号冲突的问题。为了做到这一点，它变成了一个表现积极的、dismax 形式的查询解析器。cross\_fields 所使用的排名函数与查询解析器所使用的方法相同，只有一个地方有重要的改动：cross\_fields 查询会在搜索之前，以逐字段的方式，临时修改搜索词的文档频率。如果单词 shatner 在参与搜索的字段中非常普通，只有一个问题字段除外，例如 directors.name，那么这个问题字段将会被特殊对待，被赋予一个较大的文档频率。虽然这种做法比起全字段方法来，其准确性有可能会低一些，但它也是在试图解决信号冲突的问题。

假设我们用的不是 dismax 查询解析器来解析字段，而是用的 cross\_fields 搜索，那么与类似下面的 dismax 有所不同：

```
(cast.name:william | directors.name:william)
(cast.name:shatner | directors.name:shatner)
```

我们得到的，是被 Elasticsearch 所使用的这样一种带 Blended 字样的解释语法 (Blended explain syntax)，它在我们执行搜索之前将所列字段的文档频率做了混合处理：

```
Blended(cast.name:william, directors.name:william)
Blended(cast.name:shatner, directors.name:shatner)
```

混合所有这些字段的 cross\_fields 搜索究竟是什么样子的呢？它是一种查询类型为 cross\_fields 的多匹配查询 (multi\_match query)。cross\_fields 使用了针对字段的通用查询分析器。与 dismax 查询解析器非常类似，cross\_fields 也依然存在字段同步性的问题。如果字段之间没有共享相同的分析器，cross\_fields 将会返回一个错误。认识到这一点之后，我们知道这里的大部分字段都带有英文文本形式的分析结果，因此我们可以对这些经英语分析得到的结果 (English-language-analyzed results) 使用 cross\_fields 搜索。

基于这一点，执行 cross\_fields 搜索，得到了我们想要的搜索结果，如清单 6.14 所示。

清单 6.14 针对有效字段的 cross\_fields 搜索

```

usersSearch = 'star trek patrick stewart william shatner'
query = {
    "query": {
        "multi_match": {
            "query": usersSearch,
            "fields": ["title", "overview", "cast.name", "directors.name"],
            "type": "cross_fields",
        }
    }
}
search(query)

```

用户的查询

使用 cross field 搜索

## Results

Num	Relevance Score	Movie Title
1	1.9040859	Star Trek: Generations
2	1.6575186	Star Trek V: The Final Frontier
3	1.3508359	Star Trek: Nemesis
4	1.1206487	Star Trek: The Motion Picture
5	1.0781065	Star Trek: Insurrection

这次的结果与前面的 `_all` 字段搜索有所不同。这表明，我们无法在查询阶段精确地计算出多个字段的混合文档频率。对于这种取近似值的做法，字段只能提供有限的信息。`cross_fields` 查询只能访问到各个搜索词在某一字段上的文档频率。然后，它必须尝试将这些无法保证绝对正确的值以合理的方式合并在一起。假设 `shatner` 在 `cast.name` 和 `directors.name` 中的文档频率分别是 16 和 1。基于这两个事实，我们并不能判断出，单词 `shatner` 在 `directors.name` 中出现过一次，就一定意味着 `William Shatner` 在其执导的影片中会以演员的身份出现，或者不出现。在前面的例子中，合并后的文档频率是 16。但是对于这种情况，真正的文档频率应该是两个字段的文档频率之和：17。而 `cross_fields` 方法则采取了一种稳妥的策略，它从两个字段的文档频率中，选择了最大值：16。

“自定义全字段”对字段进行物理上的合并，能够捕获到更为精确的文档频率。合并而成的 `people.name` 字段则会精确地包含搜索词 `shatner` 的一次匹配，无论他是作为导演还是演员。但是这种精确性也是要付出代价的。所有字段都要在索引阶段一次性构建完成。我们无法在搜索阶段再去合并其他的字段。而且，这种方法对存储空间的要求有时是不可忍受的，尤其在大规模系统中更是如此。`cross_fields` 尽管不是那么精确，但是它在对字段进行组合的时候允许更有针对性的灵活处理能力。不仅如此，`cross_fields` 本质上讲是一种逐词进行的 `dismax` 查询——与我们前面看到的查询解析器并无二致。因此，那些调优选项，如字段的放大系数，

以及 `tie_breaker` 参数，对我们而言仍然是适用的！不幸的是，字段同步性的问题也依然会存在。

## 6.5 结合以字段为中心和以词为中心的策略：鱼与熊掌兼得

我们已经知道了，以字段为中心的搜索经常会忽略用户基本的搜索预期。然而以词为中心的搜索，因为字段的同步性，也有其自身的问题。它对底层字段有严格的限制。因此，只支持那些基本的、不太复杂的相关性功能。回顾一下：我们在第4章中利用分析器（`analyzers`）来构造像雪花一样独一无二的一个个字段，能够对任何可以被分词的内容进行建模；在第5章中我们学习了如何将这些字段用于以字段为中心的搜索。而到了这里，我们明白了，所有这一切与以词为中心的搜索都是不相兼容的。

遗憾的是，这里并没有什么“银弹”。学会如何将两种方法结合在一起，是一个持续不断的探索过程。在这一节里，我们会谈到一些策略，以尽量平衡两种技术之间的优势，得到既满意又漂亮的相关性解决方案。大家会发现，如何将两种搜索策略结合在一起使用，完全取决于我们在数据和用户之间做出怎样的妥协。努力把握好平衡，是作为相关性技术工程师的我们，不断持续的调优工作中非常重要的一环。

### 6.5.1 将“相似字段”分到一组

在本节中，我们将对一种策略进行考察，它可以把分别以字段和词为中心两种方法结合起来，即：结合以词为中心的方法，将相似或相近的字段分组到一起。这些分组会将其自身与外部以字段为中心的搜索相结合。巧妙地对字段进行分组，可以获得“恰到好处”的以词为中心的效果，对于那些容易产生“白化象”和信息冲突问题的字段，当其拥有更多匹配的时候，我们就给予其更高的优先级，这样做不失为一种好办法，可以规避我们在使用以字段为中心的搜索时遇到的那些问题。

注意，如果用户的搜索词只会匹配一个字段，那就不存在以字段为中心的搜索遇到的那些问题。如果搜索词总是能找到自己理想的字段，那我们就不会遇到“白化象”和信号冲突的问题。例如，对于“`Star Trek Patrick Stewart William Shatner`”而言，

如果能够保证, `star trek` 只会与一个字段(我们称其为 `text`) 相匹配, 而与名字相关的搜索词, 诸如 `patrick stewart` 和 `william shatner`, 它们只会与一个涉及 `people` 的字段相匹配, 那么以字段为中心进行搜索是没有问题的。

我们将这些分组称为相似字段 (`like fields`)。正如此前所讨论的, 源数据模型本身并不具备明确的分组。我们的工作就是用这样的方式为信号进行建模, 得到分组字段, 以减少“白化象”和信号冲突对最终解决方案的影响。我们工作的目标还包括将信号匹配到用户的一般性搜索预期。通过对相似字段进行分组, 我们会更加接近于用户所关心的, 位于较高层面的信号 (`higher-level signals`)。

例如, 对于“星际迷航”的排名, 一种用户可能会想到的方法是, 将相似字段分组到一起, 标明理想的文档

- 应该 (SHOULD) 匹配搜索字符串中提到的人员 (`people`, 内部以词为中心 `people:term1 people term2...`)。
- 应该 (SHOULD) 匹配搜索字符串中提到的文本 (`text`, 内部以词为中心 `text:term1 text:term2...`)。

基于这一规则, 我们可以告诉 `Elasticsearch`, 针对“星际迷航”的搜索, 什么是理想的文档。如清单 6.15 所示, 其中有一个简单的改进, 即: 利用自定义全字段 `people.name` 作为 `most_fields` 搜索的一部分, 使我们对 `people` 的分组与前面所定义的规则更加匹配了。在清单 6.15 中大家会注意到, 为了演示基本思路, 我们只搜索了非双联词字段 (`non-bigrammed field`)。对双联词字段的搜索可能会得到更加精确的结果。

清单 6.15 对合并得到的、以词为中心的全字段 (`people.name`), 以及其他字段进行搜索

```
usersSearch = "star trek patrick stewart william shatner"
query = {
  "query": {
    "multi_match": {
      "query": usersSearch,
      "fields": ["title", "overview", "people.name"],
      "type": "most_fields",
    }
  }
}
search(query)
```

用户的  
查询

搜索非双联词字  
段 `people.name`

Num	Relevance Score	Movie Title
1	0.7104292	Star Trek: Generations



2	0.5998383	Star Trek IV: The Voyage Home
3	0.50374436	Star Trek: Nemesis
4	0.35599363	Star Trek
5	0.3373023	Star Trek: The Motion Picture

这是一种进步。通过对 `people` 字段和其他 `people` 类字段 (`directors.name + cast.name → people.name`) 进行分组, 我们避免了一类问题发生的可能性。现在, 我们能够以字段为中心进行搜索, 同时还具备了基于自定义全字段的、以词为中心的效果。我们将各种 `people` 相关的字段细心地合并成一个能够提供特定信号的字段。

## 6.5.2 理解相似字段的局限

那么, 这就是我们想要的吗? 这就是搜索问题的解决方案吗? 对相似字段进行分组是平衡两种搜索方式的正确方法吗? 遗憾的是, 在现实中, 搜索词的确会毫无逻辑地出现在一些预料之外的字段里。因此, 对相似字段分组并不总是能够包治百病。正如我们所说的, 很遗憾, 并不存在什么“银弹”, 能够解决以字段为中心和以词为中心的两种搜索方法彼此间“阴阳失衡”的问题。就像大部分编程问题一样, 我们唯有基于数据的本质和用户的需求, 仔细揣摩, 寻求折中。

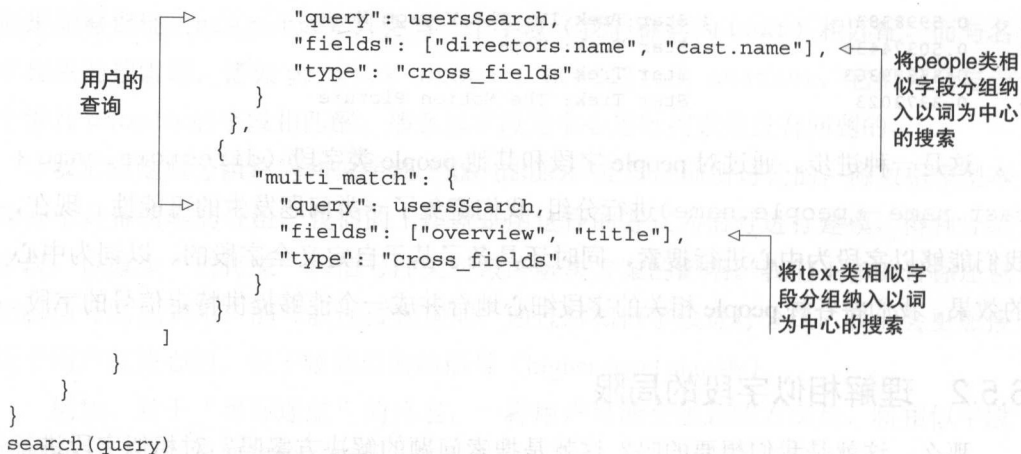
我们可以对策略进行修改, 以反映现实状况。例如, 在影片搜索中, 演员的名字出现在影片的描述中。有可能名字叫作 `Star`, 甚至带上 `Trek` 的人会得到极高的评价, 因为它们作为人名是如此稀缺; 同样, 像 `William` 这样的名字也可能会出现在标题 (`title`) 或者简介 (`overview`) 的文本中。所以, 对相似字段进行分组也许并不一定可行。

下面让我们来证明一下这个问题。此处, 我们使用 `cross_fields` 搜索, 而非自定义全字段来分别对 `text` 字段和 `people` 字段进行分组。这是我们之前所描述的策略。回想一下, `most_fields` 期望搜索的是一组字段, 而不是一组查询。因此, 要想通过内嵌 `cross_fields` 查询的方式来使用 `most_fields`, 我们就需要自己来实现 `most_fields` 的评价行为。清单 6.16 中的外层布尔查询正是这样做的。

清单 6.16 搜索由两个字段构成的分组——`people` 和 `text`

```
usersSearch = "star trek patrick stewart william shatner"
query = {
  "query": {
    "bool": {
      "should": [
        {
          "multi_match": {
```

布尔型的SHOULD子句,  
复制most\_fields的行为



```

{
  "query": usersSearch,
  "fields": ["directors.name", "cast.name"],
  "type": "cross_fields"
},
{
  "multi_match": {
    "query": usersSearch,
    "fields": ["overview", "title"],
    "type": "cross_fields"
  }
}
]
}
}
search(query)

```

Num	Relevance Score	Movie Title
1	1.1444862	Star Trek IV: The Voyage Home
2	0.75206727	Star Trek: Nemesis
3	0.7318188	Star Trek V: The Final Frontier
4	0.72360706	Star Trek: Generations
5	0.5002059	Star Trek

遗憾的是,这并没有达到我们想要的效果。由于某种原因,*Star Trek IV* (中文名《星际迷航4: 抢救未来》) 在排名中居于首位。这是为什么呢? 如果检查一下解释信息, 我们会惊奇地发现, *william shatner* 在 *Star Trek IV* 的 *overview* 字段中被提到了:

```

0.18140785, max of:
  0.18140785, weight(overview:william in 474)
0.34648207, max of:
  0.34648207, weight(overview:shatner in 474)

```

正如我们所怀疑的那样, 人的名字经常会在很多字段中出现。所以, 通常我们无法将字段清晰地划分成相似字段。由于这种混乱的、非结构化的数据集, “搜索词不是在集合 A 里, 就是在集合 B 里” 这样的想法并不总是成立的。而且, 当有更多的搜索词得到匹配时, 因为基于文本的 *cross\_fields* 查询会存在较大的偏差, 像 *William Shatner* 或者 *Patrick Stewart* 这样的名字确实与这些字段相匹配的事实, 会进一步放大对非预期匹配的偏差。

### 6.5.3 将贪婪的简单搜索和保守的放大器结合起来

我们说过, 以词为中心的搜索常常能满足我们的需要, 但很少能做到尽善尽美。

将以词为中心和以字段为中心的方法结合起来的另一种策略，是在以词为中心的基础之上进行相关性处理。以此为基础，我们可以得到更高明的、有区分度的逐字段信号（per-field signals），从而以我们认为用户会满意的方式，对文档加以放大。我们已经知道如何对字段进行加工，当这些字段得到匹配时，一定能满足用户的某些查询条件——例如，像“the user is searching for a person”或者“the user is searching for a location”这样的查询条件。根据该查询条件，我们可以认为在 TMDb 的例子中，理想的文档应当如下所示。

理想的文档

- 应该（SHOULD）让全部的搜索词与被搜索字段的超集（superset）相匹配。
- 应该（SHOULD）让针对名字的搜索与影片相关人员（people）相匹配。

我们将第一个 SHOULD 用作对文本的评价基准，它根据所有在其他子句中出现的字段，立足于由这些字段的纯文本形式所构成的一个超集。这种匹配十分贪婪；第二个 SHOULD 子句则好了很多。它有很高的区分度，且为了获得有针对性的信号，它只搜索来自第一个子句的、被分组为相似字段（如人员、场所或事物）的一个子集。

该方法有两个重要的因素：

- 任何与第二个具有高区分度特征的字句相匹配的文档，也必须与第一个具有“贪婪”特征的字句相匹配。这样一来，每一篇接受考察的文档就都有一个基准评价了。
- 不具备贪婪特征的字句必须是偏于保守的高质量信号才行，这样可以避免由于某些意料之外的匹配而掩盖了基准评价。

凭借广泛适用的基准评价和精挑细选的差别放大器（discriminating amplifiers），我们很有希望能够达到最低限度的用户满意度。随着其他次要信号的改进，我们可以继续放心大胆地将它们作为放大器——产生其他具有高区分度的信号。次要信号偏于保守的主要原因在于：如果它们不够高明，或者让一些意料之外的匹配混入其中，比如与人名无关的匹配（non-name match），那就有可能会掩盖“匹配用户所有搜索词”这一基准假设，造成一些非常低级的错误；但是，如果我们忽略次要信号，没有得到有关“人名为 William Shatner”这样的精确匹配，那么至少还有基本的以词为中心的搜索作为保底，因此情况不会太过糟糕。

让我们来做一下试验，针对每个字段的纯文本形式，利用基本的以词为中心的

cross\_fields 搜索，再加上一个额外的 SHOULD 子句，得到与人名类字段的一个子集相匹配的文档，利用双联词分析器对其进行建模，如清单 6.17 所示。

清单 6.17 具有贪婪特征的以词为中心的搜索，结合具有高区分度的相似字段

```

usersSearch = "star trek patrick stewart william shatner"
query = {
  "query": {
    "bool": {
      "should": [
        {
          "multi_match": {
            "query": usersSearch,
            "fields": ["directors.name.bigrammed",
                      "cast.name.bigrammed"],
            "type": "cross_fields"
          }
        },
        {
          "multi_match": {
            "query": usersSearch,
            "fields": ["overview", "title",
                      "directors.name", "cast.name"],
            "type": "cross_fields"
          }
        }
      ]
    }
  }
}
search(query)
1      1.6669365      Star Trek: Generations
2      1.5123603      Star Trek V: The Final Frontier
3      1.0779369      Star Trek: Nemesis
4      0.9057324      Star Trek: The Motion Picture
5      0.8793935      Star Trek: Insurrection

```

用户的查询

外层的布尔查询偏重于同时满足普通纯文本和人名类双联词字段的匹配

通过具有高区分度的双联词相似字段对人名进行搜索

针对所有被搜索字段计算纯文本的基准评价价值

现在我们已经取得了一定的进展！还需要进一步细化，但这是一个良好的开始。对 *Star Trek V* 的评价与对 *Star Trek: Generations* 的评价非常相近。这是因为双联词“william shatner”的词频是 2 的缘故吗（我们知道他既是演员又是导演）？实际上，用户根本不关心 William Shatner 是否被提到过两次！我们可以通过增强人与字段之间的关联程度来进一步提高信号的质量。我们也可以禁用词频来得到更加理想的信号（或者就像我们将在第 7 章中看到的那样，利用 constant\_score 查询来完全消除  $TF \times IDF$  的影响）。

还有一个问题，作为基准的以词为中心的搜索，应该在多大程度上成为评价的

基础呢？相比于那些更为具体的信号，以词为中心的搜索应该对最终的评价产生多大的影响呢？那些不怎么具体的，以词为中心的子句，对评价又有哪些影响呢？增加以字段为中心的 SHOULD 子句，我们会重蹈覆辙，再次遇到之前的“白化象”问题吗？即：存在大量非基础性的、以字段为中心的、雪花般特殊的字段匹配，而并非所有的搜索词在以词为中心的搜索中都得到了匹配。是的，这是有可能的。所以，持续不断地让以词为中心和以字段为中心的两种搜索方法保持彼此之间的“阴阳平衡”，会花费相关性技术工程师大量的时间。不过幸运的是，在 Elasticsearch 中，每一条查询都可以被放大处理，可以将总体评价按照我们所期望的方向进行调节，不论是向着“阴”面还是向着“阳”面。

#### 6.5.4 以词为中心与以字段为中心，查准率与查全率

上一节中提到的这种方式，让人不禁想起了我们曾经在查准率（precision）和查全率（recall）之间产生的纠结。正如我们在第 4 章中学到的，较高的查全率保证了所有正确的匹配都会出现在搜索结果中，而较高的查准率则保证了搜索结果中只包含较少的错误匹配（false-positive matches）。我们可以修改查准率和查全率的定义，只考虑显示在搜索结果第一页的前  $N$  项（比如前 10 项）结果。实际上，这对于我们得到正确的结果是非常重要的！

贪婪的、以词为中心的搜索带来了高查全率。它通过广撒网的方式保证我们捕获到所有正确的搜索结果。搜索返回的第一页中包含了一系列可能的相关性结果——很可能只是一些相当简单的匹配项的集合，这合情合理，但算不上有多高明。除此之外，用户为了得到想要的匹配，可能就得不时地在搜索结果中逐页浏览一番。加入具有高区分度的、以字段为中心的信号，提高了搜索结果的查全率。它会逐步将更多有望匹配的备选项放入第一页中，仅当其精确满足某些查询条件的时候。

6.4 节中介绍的方式，其绝妙之处在于，它允许我们自己控制想要使用多少以词为中心的成分，或者以字段为中心的成分。如果满足查准率更为重要，那么我们就可以利用放大处理（boost）来对某些特定的匹配进行微调，使其评价更容易获得提升。如果查准率并不十分重要，那么对以字段为中心的信号进行放大处理，在重要性上就会远小于对查全率的关注。

### 6.5.5 考虑过滤、放大，以及重新排名

尽管前面提到的方式是一个良好的开端，但是对于 Query DSL，人们总有一些新颖而有趣的用法。大家在第7章将会看到，我们如何精心构造信号，准确无误地利用其对排名做更进一步的控制。例如，在搜索结果中找到与一组最小数量的查询条件不相匹配的结果，将其过滤出来，也许我们可以通过这样的方式，对前面讨论过的作为基准的、以词为中心的方式进行微调；或者找到一种方法，对搜索词匹配某个基本分类，并唤起以字段为中心的信号这样的情形，统计满足此类条件的搜索词数量，更多地将其作为一种内部判断条件，决定是否重新进行排名。通常，以字段为中心的方法凭借某个特定的信号，允许我们对内容进行准确的过滤或者放大。也许我们希望明确地对距离用户较近的餐厅给予放大，或者对距离较远的餐厅加以过滤。首先，我们手里可能会有一个一般性的相关性评价。然后，再利用更为细致的建模信号，对相关性评价进行精雕细琢。

## 6.6 本章小结

- 由于“白化象”问题，以字段为中心的搜索无法满足用户对相关性相对朴素的认知，这种认知依赖于这样的一个前提，即：匹配搜索词越多的文档，其相关性应该越高。
- 信号冲突会导致异常的排名结果，因为搜索引擎将文档分解成了许多独立评价的字段。
- 以词为中心的搜索，其评价倾向于用户对相关性的朴素预期（搜索词匹配越多的文档优先级越高）。
- 以词为中心的查询解析器，比如 `query_string`，或者 Solr 的 `edismax`，解决了“白化象”的问题，但并没有解决信号冲突的问题。
- 自定义全字段的建立很好地解决了“白化象”问题和信号冲突问题，但是它增加了索引的大小。
- 混合了多字段匹配的、以词为中心的搜索方法，在查询阶段解决了“白化象”问题和信号冲突问题，但是不如自定义全字段方法那么精确。
- 以词为中心的搜索要求在字段被搜索之前就要对搜索字符串进行分析。这样做导致我们无法对各字段进行特殊化处理，以实现对各种不同的更为智能的



信号加以度量。

- 一般性的以词为中心的评价和更为智能的以字段为中心的评价，在这两者之间求得平衡，需要我们细心对待。
- 有些策略可以充分利用以字段为中心和以词为中心的两种搜索方法各自的优势（包括相似字段和具有高区分度的放大处理）。
- 对于以词为中心和以字段为中心的搜索，看待这两者的另一种不同的方式是，前者专注于高查全率，而后者则是获得更高查准率的一种工具。

# 7 调整相关性函数

## 本章要点

- 通过放大处理突出相关性内容
- 了解何时使用不同形式的放大处理
- 对热门或最新内容的排名做出改进
- 从搜索结果中过滤掉无关内容或干扰因素
- 鼓励大家创新性地使用搜索引擎的查询功能

作为一名相关性技术工程师，我们会对用户的搜索体验加以定制，以满足他们许多潜在的排名预期。分析、理解并最终实现这些搜索预期是我们工作的主要组成部分。例如，几乎每个人都觉得，新闻搜索应该列出时事要闻的最新报道；或者，餐厅搜索不但应该考虑用户的查询，而且要考虑用户与餐厅的距离远近。搜索除了文本匹配之外可能还会有其特别的排名需求。例如，如果要构建一个本地的新闻搜索，我们该怎么做呢？是否应该同时关注距离和时效呢？或者，如果要给坐飞机满世界跑的商务人士准备一个全球的餐馆搜索，那又该怎么做呢？我们是否应该将注意力放到那些建有大型机场的城市呢？

本书的前面几章已经讲述了如何为常见形式的搜索产生一般性的搜索结果。本章我们将根据用户独特的查询条件，对排名函数进行裁剪、加工和调整。大家将会

看到，我们可以成为掌握搜索引擎 Query DSL 技术的高手。为了实现对搜索的精心放大、过滤、重新排名，以及准确的排序，以满足用户独特的排名需求，我们将真正地搜索排名的方方面面进行编程。

本章是专门讨论搜索相关性的，但是我们的目标不是要事无巨细地教会大家所有的搜索技巧。每一条查询，其不同选项的排列组合实在太多了！相反，我们想教给大家的是思考问题的方法。我们将看到，使用搜索引擎的 Query DSL 实际上是要进行编程的。它不只是调节一下个别“按钮”和“转盘”就可以了。它是一门定制搜索排名的语言——新的技术层出不穷！千真万确，如果我们现在正在拜读你的博客或者著述，正从你那里学习未来几年里的一项新兴技术，那么我们一定会将其纳入本章的讨论范围。

## 7.1 何谓评价调整

利用评价调整 (score shaping)，如图 7.1 所示，相关性开始真正展现出编程的特征。在 Query DSL 中，我们通过各种工具对排名函数进行调整，以使其更加接近用户的需求。要在 Query DSL 中针对排名进行编程，最为重要的工具是放大 (boosting) 和过滤 (filtering)。在前面几章中，我们已经或多或少地见过这些术语了，不过在本书中，这些术语显得更为重要。

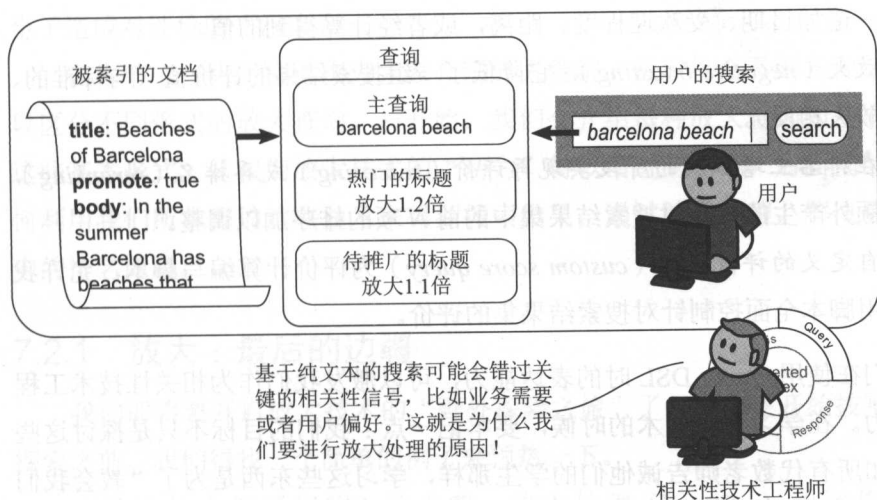


图 7.1 相关性技术工程师使用评价调整技术精心设计排名，由此得到自定义的、针对具体应用的排名规则。

现在，让我们给出严格的定义：

- 放大 (*boosting*) —— 给定的搜索结果的一个基本集合，通过放大，增加其某个子集的相关性评价。
- 过滤 (*filtering*) —— 就可能的搜索结果给定整个文档集合，通过指定过滤条件，去掉一部分文档子集。

针对放大和过滤，定义一个搜索结果的子集，并决定如何通过放大和过滤来修改相关性排名，这扩展了我们在前一章中学到的技术：

- 信号 (*signals*) —— 信号用于在查询阶段对重要的排名规则进行度量。在本章中，信号得到了进一步量化：文章发表多久了？餐馆离我们有多远？信号的出现指明了何时进行过滤或放大；信号的强弱也许决定了放大因子的取值。
- 排名函数 (*ranking function*) —— 过滤和放大直接对排名函数进行调整。例如，我们可以将一个比例因子作为乘数用于对影片发行时间的放大；而一个过滤器则可以针对排名函数所要处理的搜索结果，限定其返回的子集。

尽管放大和过滤是我们在本章所要讨论的核心技术，但是还有其他一些方法可以对评价做出调整。这些方法包括：

- 修改排序标准，使其不只局限于按相关性评价进行排序，还可以基于其他因素——诸如日期、受欢迎程度、距离，或者经计算得到的值。
- 反向放大 (*negative boosting*)，它降低了一组搜索结果的评价值（与标准的、正向放大相反）。
- 通过在排名上增加新的阶段实现再评价 (*rescoring*) 或再排名 (*reranking*)，利用额外产生的信号对搜索结果集中的前  $N$  项的排序加以调整。
- 通过自定义的评价查询 (*custom score query*) 为评价计算编写脚本，允许我们利用脚本全面控制针对搜索结果集的评价。

提高我们在使用 Query DSL 时的表达能力，可以激发我们作为相关性技术工程师的创新力。在学习这些技术的时候，要牢记一点：我们的目标不只是探讨这些技术，而是如所有代数老师告诫他们的学生那样，学习这些东西是为了“教会我们如何思考”。

## 7.2 放大：通过突出结果来实现调整

放大处理给予了我们一种权力，可以指定某篇文档相对于其他文档具有更高的相关性。仔细选择想要放大的文档并对放大产生的影响进行评估，这是相关性编程的关键技术。至于放大程度的多少，则可以基于很多规则，包括次要文本相关性评价（secondary text-relevance scores）、简单的常量，或者内容质量的度量指标（content-quality metrics），例如：受欢迎程度（popularity）、发行时间（recency of publication）或者用户评分（user ratings）。

大家可能还记得我们在前几章介绍多字段（multifield）搜索时，讨论过与各种字段相关联的放大处理。在那几章中，我们花了一定的时间，对字段匹配（例如，针对标题或正文的匹配）在多字段搜索中的相关性权重进行评估。需要澄清的一点是，那里的权重（我们称之为放大权重，boost weights）并非我们在此处所指的权重。大家在本章中将要看到的放大处理不只是针对权重的调优。在这里，我们还将使用二级查询（secondary queries），或者放大查询（boost queries）来对整体的排名函数做出修改。

为了让放大处理真正反映重要的、有意义的相关性信号，前面几章讲到的所有技术在这里仍然具有重要的意义。毕竟，如果没有好的信号可供放大，不但会影响我们对评价的调整，甚至还会因为引入干扰信号，将意料之外的结果排到前面，以至于造成对评价的“污染”。

在本节中，我们将首次为大家介绍放大查询。我们将会看到各种细分规则，用以区分不同形式的放大查询。一开始，我们会介绍每种放大查询的基本形式。了解这些基本形式，是我们放大技术的“功力”得以增进的基础。后面几节则展示了如何利用我们所掌握的这些技术来得到搜索结果，通过放大处理，解决更多现实生活中的排名问题。

### 7.2.1 放大：最后的边疆

我们即将要开启放大技术的“野外探险之旅”了，但是在开始披荆斩棘、开拓探索之前，我们得找一个简单的例子来预热一下。

应该选什么例子好呢？这个嘛，放大处理（boosting）和评价调整（score shaping）有点类似于“星际迷航”里的“最后的边疆”——它们总是时刻准备着新

的探索！因此，让我们回到前几章讨论的“星际”迷们的电影搜索，以此为例也是合乎情理的。在“星际迷航”的搜索中，有一种情况我们还没有考虑过，那就是用户在搜索时没有带“Star Trek”。粉丝们只是搜索“William Shatner”，如果电视节目 *TJ Hooker* 出现在了搜索结果的最前面，那用户就会感到不爽。让我们对电影片名“Star Trek”进行放大，将其排到搜索结果的最前面，以对放大处理的基本过程有一个初步的认识。

首先，我们需要一个基准查询（base query）。在前一章，我们已经知道了，`cross_fields` 搜索是如何为相关性技术建立起一个好的开端的，因而此处，就让我们也以它为出发点吧。在清单 7.1 中，我们搜索的是“William Shatner Patrick Stewart”，寻找的是由他们两位共同主演的影片。

#### 清单 7.1 即将进行放大处理的基准查询

```
usersSearch = "william shatner patrick stewart"
query = {
  "query": {
    "multi_match": {
      "query": usersSearch,
      "fields": ["overview", "title",
                 "directors.name", "cast.name"],
      "type": "cross_fields"
    }
  }
}
search(query)
```

← 用户的  
查询

Results

Num	Relevance Score	Movie Title
1	0.79947156	Star Trek V: The Final Frontier
2	0.67931885	Star Trek: Generations
3	0.4375222	The Wild
4	0.38154808	Dark Skies
5	0.32485005	Showtime

此处得到的结果不太令人满意。*Star Trek: Generations* 应该排在第一位（回忆一下在第 6 章中，我们对这一结果做了进一步微调）。但是，我们此刻关注的焦点并不在这里。对“星际”迷来说，最后三条结果应该都属于“星际迷航”系列电影。我们需要对这些影片进行放大才行！那就让我们来研究一下，有哪些可选的方案吧。



### 7.2.2 放大时——选择加法运算还是乘法运算，布尔查询还是函数查询？

我们已经发现了一个排名的小问题，需要进行放大处理。是时候调整排名函数，以体现我们的意愿了！从哪里开始呢？在决定如何对这些“星际迷航”的片名匹配进行准确的放大之前，让我们先来了解一下放大技术的相关知识吧。

特别是，当进行放大处理时，大家会发现，我们需要做出两个关键的决策：

- 放大处理所用的数学计算方法——在基准查询之上对评价的放大，我们是用加法还是乘法呢？
- 用于放大处理的查询——有些查询非常适合用于放大，那么我们究竟应该选择哪些查询呢？

让我们先来讨论一下数学计算方法。一篇与放大查询（本例中即“星际迷航”电影）相匹配的文档，其评价价值将会是与基准查询的评价价值组合之后的结果。选择加法或乘法，分别会得到什么样的结果呢？

- 加法放大是在基准查询的基础之上叠加放大的效果。为了达到效果，放大必须对最终的计算结果施以恰到好处的影响才行。将一个 0.01 的放大值加到评价值为 4 的基准查询上，其所产生的效果微乎其微。而将一个 100 的放大值加到该基准查询上，则会对基准查询形成极大的影响。
- 乘法放大会成倍地影响基准查询的效果。无论基准查询的评价如何，将 1.2 作为一个简单的放大系数，可以保证被放大的文档要比没有放大的文档加分 20%。

图 7.2 展示了这一选择的两种不同的含义。使用乘法放大时，我们可以将文档想象成一个气球。乘法放大会使被放大的气球“膨胀”到适当的大小。而加法放大则促使我们考虑，叠加多少放大量才能对最终的结果产生影响。

第二个重要的决策，是选择放大处理所使用的查询方法。它决定了排名函数的形式，以及我们即将用到的工具，我们可以利用这些工具对排名函数加以改造，以适应我们的需求。

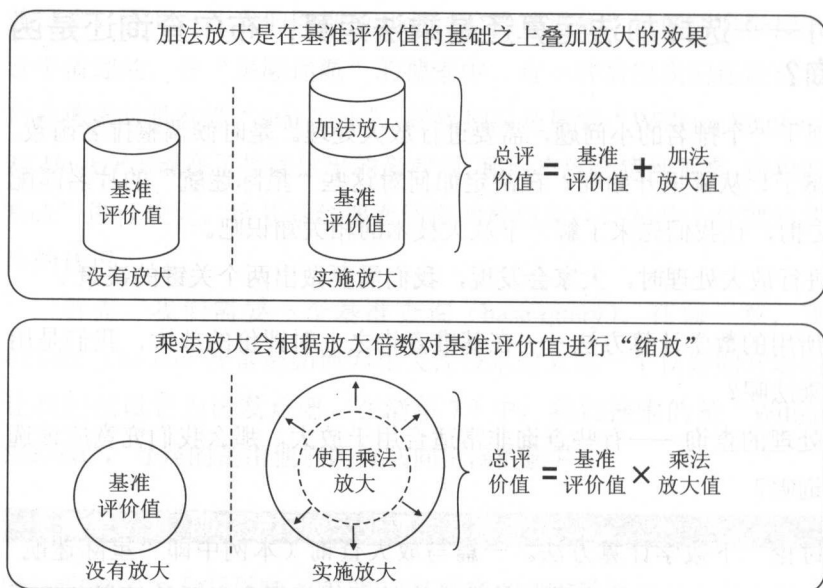


图 7.2 加法放大叠加额外的规则；乘法放大则通过乘法运算进行放大或缩小。

在 Elasticsearch 中，有两种查询可用于放大：

- 布尔查询 (Boolean query)，利用 Elasticsearch 的 `bool` 查询，通过在基准查询之上附加额外的布尔子句进行放大。
- 函数查询 (function query)，利用 Elasticsearch 的 `function_score` 查询，通过对排名函数的直接修改进行放大。

通过使用布尔查询，系统为我们做了很大程度的抽象。增加一个放大处理，意味着在基准查询的基础之上增加一个简单的 `SHOULD` 子句，并对权重进行调节。其排名算法是固定不变的（布尔查询总是加法放大），并且还附带了若干因子以供我们进行调节。

函数查询给了我们直接控制排名函数的权力。我们可以以任意形式的数学手段将基准查询与任何放大查询组合在一起。它不像布尔查询那样有固定的“公式”。因此，函数查询所用的数学算法并不能被清晰地归结为乘法或是加法。函数查询根本就是数学运算！由于函数查询是更为接近底层的控制手段，所以它能完成许多其他查询做不了的事情。

重新回到我们所讨论的问题。说到哪儿了呢？哈，对！我们有一个重要的搜索问题亟待解决，一大群“星际”迷们随时会找上门来。我们最好快点拿出解决方案来！

我们是打开第一扇门，通过基于加法运算的布尔子句来解决问题呢？还是打开第二扇门，利用函数查询来解决问题呢？那就让我们来看一下，这两种选择各自得到的结果是什么吧。

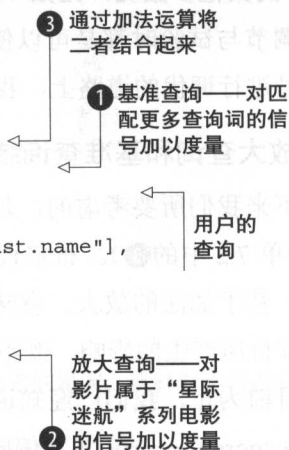
### 7.2.3 选择第一扇门：利用布尔查询进行加法放大

选择第一扇门：布尔查询。为了满足“星际”迷们的要求，我们将在 `cross_fields` 查询的基础之上利用基于加法的布尔子句进行放大处理。这样做会令搜索引擎优先选择片名中包含 `star trek` 短语的文档。大家将会看到，我们是如何将放大处理作为一种信号，用来对评价进行优化的。而且，恰当地结合用于放大的信号（`boost signal`），要求我们能够熟练掌握支持布尔查询的底层加法运算。

我们来看一下对布尔查询的放大处理是什么样子的。一条简单的 `match_phrase` 查询，涉及了在片名中匹配“`Star Trek`”，这是我们衡量“一部影片是否属于‘星际迷航’系列电影”这一信号的首次尝试。将这一查询与基准查询一起纳入一条 `bool` 查询进行放大，如清单 7.2 所示。

清单 7.2 通过基于加法的布尔子句进行放大

```
usersSearch = "william shatner patrick stewart"
query = {
  "query": {
    "bool": {
      "should": [
        {
          "multi_match": {
            "query": usersSearch,
            "fields": ["overview", "title",
                      "directors.name", "cast.name"],
            "type": "cross_fields"
          }
        },
        {
          "match_phrase": {
            "title": {
              "query": "star trek",
            }
          }
        }
      ]
    }
  },
}
search(query)
```



在揭晓最终的搜索结果之前，让我们来看一下，搜索引擎在对该查询进行评价时到底做了些什么。这将准确地揭示出，我们是怎样以及在何处对排名函数进行

修改，并根据需要对其进行放大的。

### 单独对放大处理进行优化

由于使用了一条布尔查询，首先我们对放大查询和基准查询的评价计算是分开进行的（参见清单 7.2 中的①和②）。放大处理所能达到的效果完全取决于这些底层查询的表现。在本例中，我们的查询试图在度量下列两种信号：

- Boost——影片是否是一部“星际迷航”系列的电影？（参见清单 7.2 中的②。）
- Base——所有查询词是否都出现在了被搜索的字段中？（参见清单 7.2 中的①。）

这些查询是否对所需的信号准确地进行了度量呢？例如，此处的放大查询是一条基于 title 字段的短语查询，它依赖于针对短语的  $TF \times IDF$  评价。这样的评价能帮助我们衡量出一部影片是否属于“星际迷航”系列电影吗？这看起来像是一个“yes/no”类型的问题， $TF \times IDF$  是否适合用来解答此类问题呢？

那么基准查询的情况又是怎样呢？此处的 cross\_fields 搜索是正确的选择吗？所有字段都被赋予了适当的权重吗？是否还应该搜索其他的字段呢？cross\_fields 的其他参数呢，比如 tie\_breaker？关键在于，所有这些选择在我们对放大进行调节与试验时都是可以使用的。作为相关性技术工程师，在对查询所依赖的底层信号进行调优的道路上，我们是永远不会止步的。

### 将放大查询和基准查询结合起来

接下来我们所要考虑的，是如何通过布尔查询将针对这些查询的评价组合起来（参见清单 7.2 中的③）。布尔查询的工作不只是将组成查询的各个子句在评价上累加求和。基于加法的放大，意味着我们需要仔细地叠加放大所产生的效果，对基准查询的评价所产生的影响，既不能过多，也不能过少。

到目前为止，我们已经知道了布尔查询不只是简单地相加。我们还需要考虑协调因子（coord）——它尤其偏向那些能匹配所有子句（此处，放大查询和基准查询都包括）的文档。如果不需要这样的偏向，我们可以通过 disable\_coord 选项禁用此功能。如果放大查询能够带来的额外好处已经超出了我们所要求的程度，那也许我们就会想要禁用这一功能。

最终，结合考虑了所有这些因素之后，我们对每个子句都进行了评价并且加以组合。让我们检查一下清单 7.2 给出的搜索结果，看一下是否与我们的搜索预期相符：

Num	Relevance Score	Movie Title
1	4.363374	Star Trek
2	4.0461645	Star Trek: Generations
3	3.7096446	Star Trek V: The Final Frontier
4	3.6913855	Star Trek: Nemesis
5	3.653065	Star Trek: The Motion Picture

毫无疑问，看上去好多了！但是，回想一下我们搜索的是“William Shatner Patrick Stewart”。搜索结果中的第一条记录并不是由 William Shatner 或 Patrick Stewart 主演的。难道是放大处理没有起作用？

对上述结果进行检查发现，看起来我们对“Star Trek”的放大处理也许太过了。大家会注意到，与期望的结果 *Star Trek: Generations* 相比，似乎我们非常倾向于长度更短的“Star Trek”匹配。这种行为是值得商榷的，它让我们开始怀疑，也许针对“Star Trek”的短语匹配子句，其  $TF \times IDF$  的相关性评价所衡量的并不是正确的信号。为什么我们会这么认为呢？大家回忆一下，在字段的标准化处理过程中， $TF \times IDF$  对于长度较短的字段具有极大的倾向性，因此对于“Star Trek”在片名上的较短匹配，系统给予了很高的评价。图 7.3 展示了每一层处理所产生的影响，“Star Trek”在片名上的放大处理对相关评价产生了过多的影响。



图 7.3 Star Trek 在片名上的放大处理太过了，我们需要调低它的影响，使针对于基准查询的相关性评价能够合理地发挥出更大的作用。

我们对信号进行了过度放大，而且这种放大所依据的是我们根本不太关心的因素！涉及放大的子句并没有对正确的信号进行度量。我们可以通过各种方法对此进行纠正。这其中最为简单的方法，就是减少放大的权重，将放大系数从 1 降低到比如 0.25 或 0.1。这样做就抑制了不靠谱的  $TF \times IDF$  评价所带来的影响。让我们来看一下，重新对放大处理的权重仔细地进行分配之后，其结果如何（这里我们只列

出经过修改的 `match_phrase` 子句)：

```
{
  "match_phrase": {
    "title": {
      "query": "star trek",
      "boost": 0.1
    }
  }
}
```

这样做让结果变得更加合理了。我们注意到，排在搜索结果前两名的文档和我们此前基于 `cross_fields` 的基准搜索得到的结果是一模一样的。但得到了有效改善的一点是，在紧随其后的匹配中，片名里都提到了“Star Trek”：

Num	Relevance Score	Movie Title
1	1.1662666	Star Trek V: The Final Frontier
2	1.0990597	Star Trek: Generations
3	0.6702043	Star Trek: Nemesis
4	0.62388283	Star Trek: The Motion Picture
5	0.6117288	Star Trek II: The Wrath of Khan

然而，这是一个脆弱的解决方案。简单地对放大权重进行调整可能只是将问题延缓到下一次搜索时才发生。另一种看起来更为合理的选择，是针对与“Star Trek”的片名匹配相关联的信号，去提高它的准确度。要做到这一点，其中一种办法是禁用针对该字段的标准化处理，以尝试平衡对“Star Trek”的评价。

然而，更为关键的问题是， $TF \times IDF$  在这里是否真的很重要。在用户看来，一条“Star Trek”的影片查询更像是“yes/no”“1/0”类型的问题。因此， $TF \times IDF$  可能并不能对这一信号进行准确度量。此处，我们先把这个问题放在一边。稍后，当我们深入讲解放大策略的时候，大家将会看到，有一种方法可以避免使用  $TF \times IDF$ ，却能得到“yes/no”的评价。

#### 7.2.4 选择第二扇门：利用数学运算进行排名的函数查询

是时候尝试另一种策略了：第二扇门，函数查询！在实际工作中，我们可能需要在相关性排名中使用一些数值型属性，例如一款产品的利润率，或者一篇文章的受欢迎程度。为了包含这些因素，我们需要直接对排名函数进行控制。函数查询赋予了我们这样的能力。利用函数查询，我们可以基于一些定量因素（比如受欢迎程度、发表日期和利润率）与其他搜索查询的组合，直接对排名函数进行定义。

在解决我们的“star trek”问题之前，先来看一个经典的例子：如何将最近发布



的新闻报道排到最前面。当用户在一个新闻网站上搜索“bad apple harvest”时，他们要找的可能是有关近期苹果欠收的消息。遗憾的是，在不做任何改动的情况下，搜索引擎并不知道新闻发布的时效是一个重要的因素。1901 年苹果欠收的报道可能很容易就会被排在 2011 年类似报道的前面！

我们需要告诉搜索引擎优先安排最近发布的新闻报道。我们需要了解用户的真实想法，通过数学的手段为他们的预期进行建模。

为了在时效性与用户所认为的相关性之间建立起关联关系，让我们从一个简单的、相对基础的公式开始，如图 7.4 所示。我们来定义一个函数， $R(m)$ ，等于  $1/m$ ，这里  $m$  是指文章发表经历的月数。将该函数用于以乘法为基础的放大处理：发表时间过去了一个月，其相关性就乘以 1，两个月就折半成  $1/2$ （相关性减半），3 个月就是  $1/3$ ，以此类推。

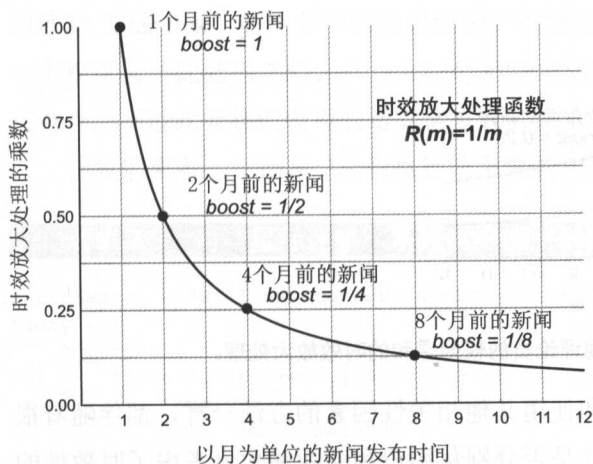


图 7.4 对新闻报道根据发布时间进行基本的放大处理

那么，和往常一样，第一版放大处理对于新闻用户就时效性这一信号在优先级方面的认知也许并不能很好地进行建模。过了 4 个月的新闻因为乘以  $1/4$  而使排名下降，这样做或许有些过于激进了。或者，还不够激进也说不定！

和使用布尔查询进行放大时的情况一样，我们所构建的信号可能并不准确。要对信号进行优化，需要调整数学运算本身，而不仅仅只是  $TF \times IDF$ ，以及其他基于文本的评价因素（也有人称之为经验系数，fudge factor，这里我们称之为评价调整，score shaping）。不过，即便我们在数学运算的自由度方面得到了增加，但要求是一样的：我们依然需要确保信号的准确性。此处有如下几个相关的问题：时效性所产

生的影响随着时间向前推移有没有下降得过快？或者说过于激进了呢？放大系数作为乘数足够大吗？是否应该使用不同的数学公式来降低排名呢？所有这些问题都关系到我们如何衡量用户对时效性可能的优先级安排。

例如，我们可能已经知道了，我们的用户都是新闻评论员。他们重视新闻时效，但偶尔也需要研究一下稍旧一点的新闻故事。所以去年苹果欠收的消息仍然是有价值的。通过某些参数的调整（和公式的改造），或许像  $R(m) = 8 / (5 + 3m)$  这样的时效函数，由于让时效性少了些激进，其效果会更好，更加贴近新闻评论员们的需求，如图 7.5 所示。

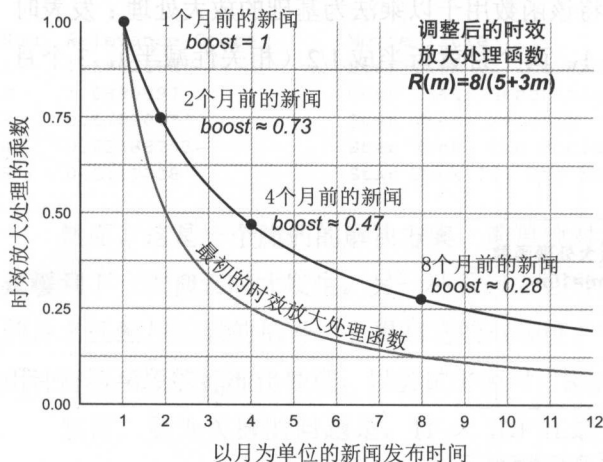


图 7.5 调整后的时效放大处理：符合新闻评论员的较为温和的时效放大处理。

使用数学方法获得正确的信号和使用其他相关性因素的方法一样，都伴随着很多相同的问题需要考虑。所用的公式是否分别从用户和商业的角度考虑了时效性的优先级呢？这与我们在处理针对布尔查询的放大时所讨论的内容是完全相同的，不同之处只不过是我们可以使用任意形式的数学方法，这让结果的可能性变得更加开放，产生了更多的决策机会。

## 7.2.5 函数查询实践：简单的乘法放大

让我们回到 Star Trek 的例子，将函数查询用于 Query DSL。在处理更为复杂的数学问题之前，我们先来看一条基本的函数查询。为了便于大家练手，我们将从解决“Star Trek”的片名放大问题着手。在本章的后续部分，我们将对此处所讨论的简单函数进行扩展，以解决规模更大、更有挑战性的问题。

在本例中，我们希望将片名中带有“Star Trek”的文档乘以值为 2.5 的因子。其所实现的函数采用的是一种简单乘法放大，用数学方式表达，则如下所示：

$$B(\text{title}) = \text{"star trek" in title?} \begin{cases} \text{no: 1} \\ \text{yes: 2.5} \end{cases}$$

该函数非常简单，当某个特定匹配发生时，就返回某个值；不匹配，则返回另一个值。

让我们开始工作吧！Elasticsearch 是用 `function_score_query` 实现函数查询的，如清单 7.3 所示。`function_score_query` 的核心部分是基准查询的 `query` 参数①——这是又一个结构完整的 Query DSL 查询。与该查询的相关性评价捆绑在一起的是包含数学运算的 `functions` ③。里面只有一个简单的函数，即我们的 `star trek` 函数②。如清单 7.3 所示，根据指定的权重和过滤条件，满足过滤条件（在本例中，即针对“star trek”在片名中的短语查询）的所有文档都将因为该函数的存在而得到一个权重值（此处为 2.5）。`function_score_query` 还允许我们指定函数以何种方式结合，不过此处我们使用的是默认方式：将所有函数与基准查询的评价值相乘。

让我们来看一下清单 7.3 的执行结果究竟如何。

清单 7.3 对“星际迷航”系列电影使用乘法运算

```
usersSearch = "william shatner patrick stewart"
```

```
query = {
  "query": {
    "function_score": {
      "query": {
        "multi_match": {
          "query": usersSearch,
          "fields": ["overview", "title",
                    "directors.name", "cast.name"],
          "type": "cross_fields"
        }
      },
      "functions": [
        {
          "weight": 2.5,
          "filter": {
            "query": {
              "match_phrase": {
                "title": "star trek"
              }
            }
          }
        }
      ]
    }
  }
}
```

```
search(query)
```

```
Results:
```

① 基准查询

用户的查询

③ 将函数应用于基准查询

乘法函数：当短语“star trek”出现在片名中时，基准评价 ② 值 × 2.5

Num	Relevance Score	Movie Title	
1	1.9986789	Star Trek V: The Final Frontier	5.4
2	1.6982971	Star Trek: Generations	6.5
3	0.6236526	Star Trek: Nemesis	6.3
4	0.60909384	Star Trek II: The Wrath of Khan	7.1
5	0.5075782	Star Trek IV: The Voyage Home	6.7

最终的结果表明这样的放大效果是合理的：“星际迷航”系列影片被排到了前面。与加法放大的例子中我们需要时刻关注针对放大查询的  $TF \times IDF$  评价是否合理有所不同，大家会注意到，此处我们并没有考虑  $TF \times IDF$  评价。从某种程度上来说，尽管权力更大，责任也更大了，但最终的结果却是变得更加简单了。

### 7.2.6 放大处理的基础：信号，处处是信号

我们已经了解了放大处理的基本形式。不论我们选择何种形式的查询，有一点是始终不变的：信号的本质力量。我们的信号建模工作是相关性的基础。在放大处理中尤其如此。对“Star Trek”的放大，我们在权重上施加了足够重要的“影响力”了吗？ $TF \times IDF$  评价是否对用户所认为的重要信息进行度量了呢？我们的查询用对了吗？数学函数用对了吗？这些因素与用户从直觉角度所认为的优先级相符吗？

我们对度量这些信息的特征和信号把握得越好，以符合人类而非搜索引擎所能理解的方式解决排名问题的能力，就会变得越强。

## 7.3 过滤：通过排除的方法对结果进行调整

调整评价的最后一个基本要素是排除不需要的搜索结果。用户常常不希望看到某些结果，我们的工作就是要保证这些结果被排除在外。精心地设计查询，指明哪些结果应该被排除在外，然后通过精雕细琢的信号，对这些排除在外的结果加以控制。这样一来，通过将一些已知类型的、不相关的结果排除在外，我们就可以增加查询结果的精度。本节将简单介绍过滤器及其在整个排名函数中所扮演的角色。

在实现用户体验的相关功能时，我们经常会想到使用过滤手段。过滤器将一部分结果从用户的考虑范围之内移除出去。我们可将其看成是对搜索的一种清理（declutter）工具。允许用户通过用户界面，手动选择过滤条件，帮助他们得到需要的相关性内容。也许用户会自己选择分类，例如在影片搜索中限定只显示 DVD，过滤掉数字流媒体或蓝光选项。或者在购买电视的时候，也许会将过滤条件设定为一

组特定的规则：50 英寸、等离子、免运费。通过过滤器引导用户得到其所需要的相关性内容是我们第 8 章将要讲述的内容。

然而过滤器不单是一种控制用户体验的东西。它就像是一扇门。一个精心设计的过滤器可以帮助我们更加精确地控制查询结果的准确度，去除那些经过明确声明的不相关的内容。

我们可以从“星际迷航”搜索中举出一个简单的例子来。在前一节中，我们关注的焦点是如何对 Star Trek 影片的搜索结果进行放大，将其排到前面。而实现这一效果的另一条途径则是将那些与“星际”影片无关的搜索结果去掉。这种方法如何实现呢？

在 Elasticsearch 里，过滤器是以布尔查询中的过滤器子句的形式表现的。在过滤器中，我们可以定义如何对文档集进行限制；在清单 7.4 中，我们通过一个短语查询（phrase query）来实现这一功能。

清单 7.4 对“星际迷航”的搜索结果进行过滤而非放大

```
usersSearch = "william shatner patrick stewart"
query = {
  "query": {
    "bool": {
      "should": [
        {
          "multi_match": {
            "query": usersSearch,
            "fields": ["overview", "title",
                      "directors.name", "cast.name"],
            "type": "cross_fields"
          }
        }
      ],
      "filter": [{
        "query": {
          "match_phrase": {
            "title": "star trek"
          }
        }
      ]
    }
  }
}
search(query)
```

用户的查询

只包含“Star Trek”的结果

如果我们不考虑评价，而只是想显示或不显示某些搜索结果，那么过滤可能是一种更为简单的解决方案。而且通过只对有限的搜索结果集进行评价，过滤可以使放大的准确性更高。从而明确地去除了无关的结果，避免了在相关性计算中出现的

一系列边界情况。

就像我们对待放大处理那样，我们也应该思考一下，如何根据信号进行过滤。我们用来进行过滤的查询，应该明确地从用户和业务这两个角度对需求进行度量。在前一节中，我们曾经对短语查询所度量的是否是正确的信号有所怀疑。这里也是一样的。如果由于某种原因，我们发现对于“这是一部‘星际系列’影片”这样的信号所用的度量方法有误，那就需要深入考虑一下所使用的这些查询了。由于要试图包含或排除各种搜索结果，因此过滤器查询有可能会和放大处理一样复杂。

在我们的职业生涯中，各种排名问题以后还会层出不穷，了解各种技术可以做到有备无患。在某些情况下，除了放大处理以外，其他的解决方案说不定会更加有效。

## 7.4 满足业务需求的评价调整策略

我们应该如何利用评价调整工具，比如放大和过滤，来解决特定的问题呢？要记住，本章讨论的是如何使 Query DSL 满足我们的意愿。但是在实际应用中，我们需要考虑的并不是自己的意愿；我们所要解决的全部都是那些从用户和业务角度出发对排名需求构成的挑战！本节将使用我们目前所学到的工具，帮助大家处理真实的业务和用户需求。

评价调整的问题，涉及人类能够理解的、以通俗易懂的英语形式表达的排名优先级，还有如何将其翻译成可以对排名函数实施控制的放大、过滤，以及查询。不懂技术的老板、内容管理者，以及其他一些同事可能会要求“把精确匹配片名的影片直接排到前面”或者“对过去五年发行的电影进行放大处理”。我们的工作就是要通过包含合适的的数据，并利用所掌握的搜索引擎的特性，将这些以英语形式表达的期望翻译成“搜索引擎的语言”。

本节我们将探讨针对搜索引擎的控制策略，以回答上述这些问题。为了对评价调整的问题进行分析，我们引入了一个框架，由两个部分共同构成。

首先，我们的框架立足于一系列高准确度的放大信号，这些信号基于用户所熟悉的问题进行排名，如：“影片有精确的片名匹配”或者“影片发行于5年以前”。为了构建这样的信号，我们会引入合适的的数据，仔细控制字段的组合，并对字段的查询和评价方法进行管理。

其次，我们的框架考虑了如何将用户的预期组合起来，形成一个更大规模的解



决方案。要覆盖原有的排名方法，一方面我们可以基于某一特定信号的强度采取一些办法，要不然就退回到基准查询。另一方面，我们也可以选择一些小幅度的，或者 tie-breaker 式的放大形式——利用这些锦上添花的低优先级的放大手段，我们可以对相关性进行上下微调。

如图 7.6 所示，这两个阶段是高度相互依赖的。



图 7.6 评价调整过程中的两种行为活动,包括信号建模(对排名信息进行度量)和排名函数设计(结合来自不同信号的影响,以实现商业逻辑)。

我们通过放大处理实现对排名的微调或者重建,有可能在多大程度上会依赖于信号的准确性呢?也许这就是一个基于文本的相关性微调而已。又或者是一个通过乘法放大实现的微调,依赖于设计良好的函数作为乘数因子。我们总是不断地考察这两个不同的层面,对信号进行优化,思考如何运用这些信号,并修改相应的排名函数。搜索是一项高度迭代化的工作。通过放大,我们可以深入地控制排名函数——这也是迭代的有趣之处。让我们通过一个真实问题的解决方案,来看一看这种方法学思想的实际应用吧。

### 7.4.1 搜索所有影片

为了彰显这个全功能搜索引擎的全部威力,我们打算换一个方向。前面我们已经为获得影片推荐的“星际”迷们开发出了如此好的一个搜索应用,我们的下一个目标是要实现一个针对所有人的影片搜索。为了更好地服务于这个更大范围的用户群体,我们将有机会使用一些更加复杂的排名规则。让我们花一点时间先来了解一下,客户要求我们实现的这个搜索方案到底是什么样的。然后再来看一个有趣而又复杂

的搜索方案，该方案中所反映的那些棘手的问题大家在实际工作中肯定都会遇到。

老板已经对我们说过了，“要实现正确的搜索，我觉得大体上你需要在搜索引擎里实现下面这些逻辑，包括：”

1. 如果用户的查询是一个完整而精确的片名匹配，比如搜索“Star Trek Generations”，那么精确匹配片名的影片就应该直接被排到搜索结果的最前面。
2. 如果用户的查询是一个完整而精确的人名匹配，比如搜索“William Shatner”，那么对于搜索结果就应该根据影片上映时间由近到远，以及评分由高到低的顺序进行排列。
3. 优先安排包含某个人全名的搜索结果，比如查询“William Shatner Star Trek”。
4. 否则，我们就针对用户和查询，采用一般意义上基于文本的相关性计算方法，来对搜索结果进行评价。

图 7.7 所示的流程图总结了上述这些规则。

电影搜索排名规则的流程图  
(从有利于业务的角度出发)

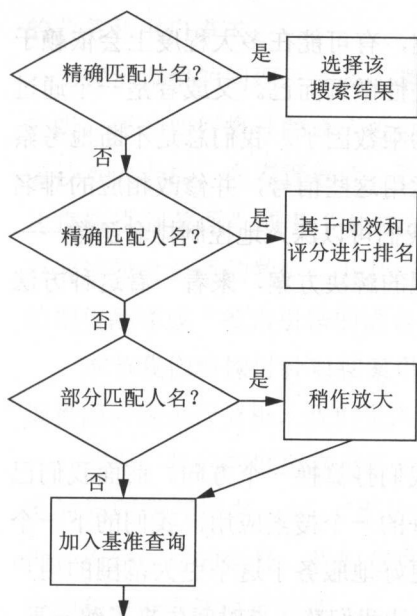


图 7.7 以流程图的形式来表达客户的业务排名规则，这些规则通过业务来指定，并能被业务所理解。

这里有相当多的内容需要解释。前两条规则说明了，以高精度水准对精确匹配进行度量是很重要的。我们需要设计出字段，能够捕获这样的排名信号。在步骤2中，实现人名的精确匹配，这一点值得注意。通过人名进行搜索时，我们有可能得到很多匹配的影片，不过老板已经提前为我们想好了，出现这种情况时应该如何对结果进行排序。老板觉得用户想看的，可能是这些演员或导演的最新、最好的作品。到目前为止，我们已经通过两章的篇幅研究了步骤3要做的工作。因此老板一定相信我们能够继续胜任后续的工作。大家将会看到，我们是如何通过步骤3在小范围内对几个匹配项的相关性评价进行放大的，如果大家还记得，我们在第6章的末尾也做了类似的事情。最后，第4步，作为基准的相关性评价，其所度量的，是有多少搜索词匹配了整篇文档。回忆一下前一章的内容，以词为中心的搜索实现的就是这个功能。

这样一个递进的顺序听起来似乎很不错，不过希望大家能够在脑海里敲响警钟。我们怎么知道这些规则是否符合业务需求呢？为什么这些规则对于我们的排名方案而言是高优先级的呢？这些问题都很重要，但是不必担心，在后面几章里，对于这些涉及精度的问题，我们将会找到相应的答案。确保正确的排名规则是非常重要的！不过，知道如何将这以纯英语形式表达的排名规则翻译成搜索引擎的语言，也是非常重要的，这正是本章的重点。

那我们就开始吧！当拿到这些规则以后，我们要做的首要步骤是看一下我们的字段是否能够实现所要求的信号。一旦确信我们可以支持这些信号——我们的字段可以反映现实情况——那就可以开始执行放大处理的第二步：设计出满足需求的排名函数。

### 7.4.2 对放大信号进行建模

作为一名相关性技术工程师，当被告知我们要根据某些条件，比如“当搜索精确匹配电影片名时”，进行放大处理，那接下来的问题就应该是：如何对其进行准确度量？这就是本节要解决的问题。我们会将人类语言表达的想法翻译成某种特定信号的实现，可以将其用于更大范围的排名函数之中。实际上，在本书中我们已经这样做过好几次了。不过，这一步骤是能否编程实现相关性排名规则的基础。如果没有可操作的信息，我们如何知道排名函数是否优先选择了正确的文档呢？

要实现业务需求，在设计整体的排名函数之前，我们需要确保自己能够正确回答出如下这些重要的问题：

- 用户的查询是一个完整的、涉及电影片名的精确匹配吗？
- 用户的查询是一个完整的、涉及演员或导演全名的精确匹配吗？
- 在用户的查询中，包含了演员或导演的全名吗？

我们的目标是要构造出具有高区分度的信号，需要让信号像挑剔的品酒师那样——小心翼翼地让每一篇文档的“精华”飘进他们那敏锐的鼻子里，最终只接受经过慎重考评的文档。我们早就已经开始了这样的流程：我们具有非常合理的与人名相匹配的字段，这些字段是基于前面几章提到的双联词（bigrams）构造的。这就涉及第三条规则——查询中的某一部分是否与影片所涉及的人员具有较高精度的匹配。我们将再次回到我们的老朋友“人名匹配（name-matching）”这一话题上来，不过先来看一下，我们是如何对其他排名规则进行度量的。

在这些信号中，有两个信号是要求精确匹配，而非部分匹配的。要实现这样的匹配，我们不能只是简单地将用户的查询文本封装成短语查询。搜索短语“Star Trek”会匹配任何名称中包含“Star Trek”的文档，包括 *Star Trek: Generations*。这不是查询和片名之间的一个精确匹配。是的，只有在查询词与电影片名中的文字完全一致时，我们才会进行放大处理。

因此，现在应该是时候展开讨论信号建模了！我们需要构造出这样一个字段，它能非常准确地度量出这些信息。聪明的读者可能在想——不就是精确匹配吗？这很容易呀！我们可以禁用分词处理（tokenization）或者利用基于关键词的分词器（keyword tokenizer），对片名字符串进行精确搜索。这样做会产生一系列精确的 token，比如 `star trek` 或者 `star trek:insurrection`。是的，这样做也许能行，但实际情况是，很少能为我们返回真正想要的结果。我们仍然需要做一些分析处理，即使在进行所谓的精确匹配时，也是如此。例如，我们可能会去掉 `star trek:insurrection` 中那个恼人的冒号，来帮助用户实现不带冒号的搜索。我们也可能需要做一些词干处理（stemming）或者任何其他形式的以逐词方式进行的标准化处理（per-term normalization）技术。

与禁用分词处理的方法有所不同，我们接下来将会采用“标记 token（sentinel tokens）”技术。标记 token 代表了文本（也许是一个句子、一个段落，或者文本的起止点）中重要的边界特征（boundary features）。在本例中，我们会将如下所示的

起止标记插入到片名文本中：

```
"SENTINEL_BEGIN Star Trek SENTINEL_END"
```

这些标记就是代表字符串起止点的 token。我们在执行搜索时，将这些标记包含在查询中，以短语查询“SENTINEL\_BEGIN <user query> SENTINEL\_END”的形式执行搜索。其结果是，仅当文档匹配短语查询，且标记出现在正确的位置时（在本例中，即当文本的开始和结束部分都匹配时），才会被认为是匹配的。

实现这一思路可以有多种方法。其中一种比较明智的做法，是为搜索引擎写一个插件。回忆一下第 4 章，token 过滤器可以在分析阶段拦截并修改 token 流。我们可以自己编写 Lucene 的 token 过滤器，用来插入边界特征。不妨找一本介绍 Solr、Elasticsearch，或者 Lucene 的权威著作，让它来告诉你怎么写插件。为了简单起见，我们将选择一种较为简单的方法，在搜索引擎之外实现标记的插入。

首先，我们将修改 TMDB 的索引函数，在代表影片片名的、我们称之为 title\_exact\_match 的一个新字段的前后分别插入标记字符。要实现这一功能，我们需要修改 TMDB 的 reindex 函数，令其在索引之前调用一个转换函数。在文档进入 Elasticsearch 之前，这将是我们的操作文档的一个小小的 hook。

```
for id, movie in movieDict.iteritems():
    ...
    esDoc = movie
    transform(esDoc)
    ...
```

接下来，我们将定义转换函数，用以创建包含标记 token 的字段，如清单 7.5 所示，我们需要这样的字段来帮助我们解决问题。

#### 清单 7.5 为精确匹配插入起止标记

```
SENTINEL_BEGIN = 'SENTINEL_BEGIN'
SENTINEL_END = 'SENTINEL_END'
def transform(esDoc):
    esDoc['title_exact_match'] = SENTINEL_BEGIN + ' ' + \
                                esDoc['title'] + ' ' + SENTINEL_END
reindex(analysisSettings, mappingSettings, movieDict)
```

估计大家一定正在满腹狐疑，这样做的效果如何呢？所有这些设置、分析、映射等手段，都适合于对这一信号的度量吗？我们有许多可选的优化手段可以考虑。

还记得默认分析器吗？它的本质是将字符串分解成 token。我们的默认设置是英文分析器。这是否合适呢？英文分析器会因为词干处理的原因而将“Stary Trek”匹配成“star trek”——这是我们期望的行为吗？也许我们希望的是尽可能地接近逐字母的精确匹配呢？我们的工作就是要努力挖掘这些细节，以确保度量的是对用户有用的排名信号。

现在，让我们来看一下，对于实现了精确匹配的前述信号，执行查询将会返回什么样的结果。搜索使用了针对这一字段的标记，是否就能为我们提供所需的信号呢？参见清单 7.6。

清单 7.6 单独测试精确匹配的信号

```
usersSearch = "star trek"
query = {
    "query": {
        "match_phrase": {
            "title_exact_match": {
                "query": SENTINEL_BEGIN + ' ' + \
                    usersSearch + ' ' + SENTINEL_END,
                "boost": 0.1
            }
        }
    }
}
search(query)
```

Num	Relevance Score	Movie Title
1	7.172676	Star Trek

带标记的  
用户查询

太好了，起作用了！我们已经明确地构造出一个精确匹配的字段。不过随着知识技能的不断积累，我们也会一直被各种问题所困扰。TF × IDF 评价在这里重要吗？我们要用它作为内容的信号吗？也许不。随着搜索技能的提高，我们也会逐步了解影响信号度量的各种因素。

最后，我们将对 name 字段重复同样的步骤。对于该字段，我们会建立一个非双联词的（non-bigrammed）、普通文本字段，和前面的 title 字段类似。

清单 7.7 精确匹配的字段 name 和 title

```
SENTINEL_BEGIN = 'SENTINEL_BEGIN'
SENTINEL_END = 'SENTINEL_END'
def transform(esDoc):
    esDoc['title_exact_match'] = SENTINEL_BEGIN + ' ' + \
        esDoc['title'] + ' ' + SENTINEL_END
    esDoc['names_exact_match'] = []
```

为精确匹配的  
title 字段添加  
标记



```
for person in esDoc['cast'] + esDoc['directors']:
    esDoc['names_exact_match'].append(SENTINEL_BEGIN + ' ' +
                                     person['name'] + ' ' +
                                     SENTINEL_END)
```

为精确匹配的name  
字段添加  
标记

现在，我们已经具备了在更大范围内处理相关性解决方案的基础条件。我们构建了相关性信号，并进行了独立测试。接下来，我们要看一看如何将 **name** 和 **title** 的精确匹配信号纳入更大规模的排名函数之中。如果对于排名函数的改进有任何方面的需求（或者想重新构造新的信号），我们总是可以从本章的内容中有所收获。

### 7.4.3 构造排名函数：增加具有较高价值的层级

一切工作就绪以后，下一步我们来探讨一下，如何将它们结合起来构成一个更大规模的排名函数。排名函数是各种信号竞相争锋的场所——希望它们能够和睦相处，不要争得你死我活！

本节向大家介绍我们最喜爱的调节技术之一：构建评价层（scoring tiers）。基于我们对放大信号中所提供信息的信心，按照层级来表示排名常常是很有用处的。对于那些为搜索对象提供明确信息的特定信号，我们不应该简单地“逐层纳入”或者“与其他因素放到一起来统筹考虑”。如果我们真的具有明确指向搜索对象的高准确度、高辨识度的信号，那么完全没有必要去考虑其他因素。相反，在该项技术中，我们会让这些具有更高价值的匹配，立足于其自身的、具有高度评价的层级，将评价价值纳入它们自己所属的类别之中。能否在那样的层级上进行评价，取决于什么才是影响搜索结果的重要因素，就比如前一小节中的精确匹配。

图 7.8 展示了两种评价价值。第一种是作为基准的、以黑色标示的评价价值。它们与基准查询相匹配。这样的查询产生了大量可能的相关性结果。在本例中，我们将再次选用本章前面提到的以词为中心的搜索。第二种则是增强了的、以白色标示的评价价值。这些都是被高度放大的结果。

我们可以利用各种放大技术实现这样一个具有高价值的评价层。清单 7.8 采用了布尔查询；我们可以看到作为第一个布尔子句的放大查询。注意，用于放大的权重③被设得极高——一直到了 1000。

为结果创建具有较高价值的层级

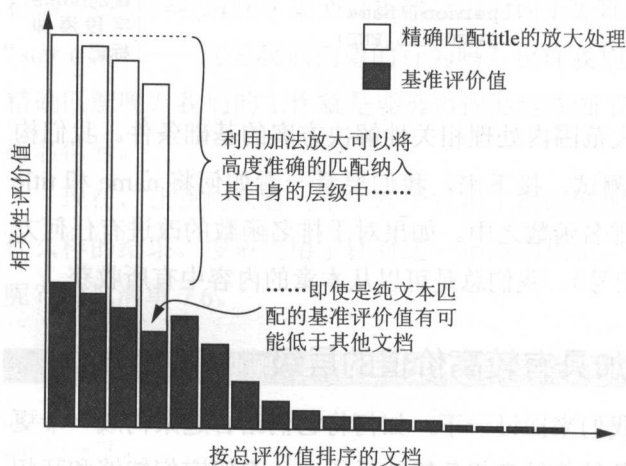


图 7.8 高精确度的层级中明确包含了更有价值的搜索结果，从而让具有高查全率、以黑色标示的基准查询得到了增强。

#### 清单 7.8 在针对片名的精确匹配之上进行的布尔放大

```
query = {
  "query": {
    "bool": {
      "disable_coord": True,
      "should": [
        { "match_phrase": {
          "title_exact_match": {
            "query": SENTINEL_BEGIN + " " + \
              usersSearch + " " + SENTINEL_END,
            "boost": 1000,
          },
        } },
        { "multi_match": {
          "query": usersSearch,
          "fields": ["overview", "title",
                    "directors.name", "cast.name"],
          "type": "cross_fields"
        } },
      ],
    },
  },
}
```

1 放大查询——精确匹配电影片名

3 高度放大产生过度影响

2 基准查询

用户的查询

```
search(query)
```

我们可以看到，“Star Trek”的搜索结果正如我们所期待的那样，直接被排到了

最前面：

Num	Relevance Score	Movie Title
1	7.1752715	Star Trek
2	0.0020790964	Star Trek: The Motion Picture
3	0.0020790964	Star Trek: Nemesis
4	0.0020790964	Star Trek: Insurrection
5	0.0020790964	Star Trek: First Contact

这是“Good Will Hunting”的搜索结果：

Num	Relevance Score	Movie Title
1	7.914943	Good Will Hunting
2	0.0016986724	The Hunt
3	0.0012753583	Good Night, and Good Luck.
4	0.0011116106	As Good as It Gets
5	0.00058992894	Saw V

此处，我们为排名函数增加了一条小小的规则，它应该非常适用于针对片名的精确匹配这样一种很常见的使用场景。大家带着问题，继续往下看，我们将逐步开始介绍放大处理中的各种专门的模式。

为“中等可信度”的放大 (medium-confidence boosts) 增加新的层级 (双联词又回来了！)

到目前为止，我们已经为更大范围内的布尔查询添加了一个层级：精确匹配。让我们再来测试另一个有一定精确度但是程度稍差一些的例子，即：搜索时将全名作为查询的一部分。这种情况要求其自身所在的层级具有一定的优先级，但是优先级没有像针对片名的精确匹配那么高。考虑这样的放大，我们需要从两方面着手：

- 我们将使用一个较小的放大值 100 建立一个中间层级。
- 我们将对双联词匹配中的信号进行优化，以确保所度量的都是重要的信息。

第二点是最为重要的。让我们从前一章介绍的基于双联词的人名匹配说起。基于字段 `cast.name.bigrammed` 和 `directors.name.bigrammed`，我们使用了 `cross_fields` 搜索。在清单 7.9 中，我们将该查询作为一种放大手段，与基准查询结合在一起使用。遗憾的是，就像下面针对“Star Trek Patrick Stewart”的搜索一样，将两者组合之后得到的查询，其执行结果并不总是尽如人意。

清单 7.9 为双联词查询增加一个子句（基准查询并未列出）

```

'query': {
  'bool': {
    'should': [
      { 'multi_match': {
        'query': usersSearch,
        'fields': ['overview', 'title',
                  'directors.name', 'cast.name'],
        'type': 'cross_fields'
      } },
      { 'multi_match': {
        'query': usersSearch,
        'fields': ['directors.name.bigrammed',
                  'cast.name.bigrammed'],
        'type': 'cross_fields',
        'boost': 100
      } }
    ]
  }
}

```

基准查询

用户的查询

针对人名匹配的中间层级

1	0.21988437	Star Trek: Insurrection	6.3	1998-12-10
2	0.21988437	Star Trek: First Contact	6.9	1996-11-21
3	0.19890885	Gnomeo & Juliet	5.9	2011-01-13
4	0.19890885	Excalibur	6.7	1981-04-10
5	0.19462639	Star Trek: Nemesis	6.3	2002-12-12

奇怪的是，*Gnomeo & Juliet*（中文名《吉诺密欧与朱丽叶》）和 *Excalibur*（中文名《黑暗时代》）的排名要比“Star Trek”的结果还要靠前。为什么这些和“Star Trek Patrick Stewart”无关的结果会优于“Star Trek”的结果呢？通过查看解释信息，我们惊讶地发现，匹配“Patrick Stewart”的 *Excalibur*，在评价上要比 *Star Trek: Nemesis*（中文名《星际旅行 10：复仇女神》）得分还要高。

以下是针对 *Excalibur* 的评价值：

```
0.5910474, weight(cast.name.bigrammed:patrick stewart in 315)
```

而 *Star Trek: Nemesis* 得到的评价则比较低：

```
0.5171665, weight(cast.name.bigrammed:patrick stewart in 631)
```

那么，为什么会这样呢？原来，针对 *Excalibur* 的匹配是由于字段被归一化处理（field norms）的原因才得到了更高的评价。还记得我们的系统倾向于内容长度更短的字段吗？*Excalibur* 中的演职人员数量更少，因此对于搜索引擎而言，其匹配显示了更强的相关性。但是用户并不关心这个；对于回答“影片是否由 Patrick Stewart 主演？”这一问题而言，这个因素根本无关紧要。

是时候引入更多信号建模的手段了。正如前面利用标记 token 一样，此处我们

需要严格控制字段的匹配和评价。我们可以重新回顾一下：定义映射信息、在禁用归一化处理的情况下执行 `reindex`，以及重新执行搜索查询这几个步骤。首先，深入层层嵌套的映射信息内部，禁掉归一化处理，并对影片数据重新执行 `reindex`：

```
mappingSettings['movie']['properties'] \
    ['cast']['properties'] \
    ['name']['fields']['bigrammed']['norms'] = {'enabled': False}
reindex(analysisSettings, mappingSettings, movieDict)
```

当再次执行搜索时，我们看到了，位于中间部分的排名变得更为合理了：

1	0.03920228	Star Trek: Insurrection	6.3	1998-12-10
2	0.03920228	Star Trek: First Contact	6.9	1996-11-21
3	0.03917096	Star Trek: Nemesis	6.3	2002-12-12
4	0.03917096	Star Trek: Generations	6.5	1994-11-17
5	0.03820324	Gnomeo & Juliet	5.9	2011-01-13

所有涉及 Patrick Stewart 的电影都被排到了前面。在这个集合中，基准查询对“Star Trek Patrick Stewart”的排名变得更加精确了。

### 为相关性制作一个分层的（tiered）夹心蛋糕

在前面的例子里，我们实现了业务所要求的部分排名规则。针对片名的精确匹配具有如此决定性的作用，以至于老板告诉我们可以忽略所有其他因素。为该匹配赋予更高的评价就体现了这一点。类似的，针对名字的匹配也是如此，只是比起片名来稍逊一筹。

由于我们对查询的优化更加贴近于用户所需要的信号，所以上述这两种放大手段的效果都是不错的。在我们的测试里，为放大查询赋予相应的放大权重并不是因为查询本身具有如此级别的优先级，而是因为相关信号具有难以置信的高准确度和高区分度。与每一个信号都匹配，就意味着我们中了相关性的“大奖”。恰好这一次，我们碰巧准确地知道了用户想要搜索的是什么，所以好好把握这些信号吧！在下一节中，我们将介绍另一个具有较高价值的层级（high-value strata），还是考察针对人名的精确匹配规则，但是我们将把重心转移到一种针对用户目标的、更具数学意义的建模方法。

#### 7.4.4 利用函数查询对具有较高价值的层级进行评价

接下来的一项重大的挑战，要求我们在为排名函数进行编程时，要进一步使出

浑身解数才行。在学习相关性编程的过程中，我们将逐步了解到一系列彼此相关的高阶技巧。在这一节里，我们将使用前一节中讲到的具有较高价值的层级，并将其与另一种技术结合起来，即：精心结合了以影片的各种特征为基础、经过反复调优的函数查询。

当存在演职人员涉及姓名的精确匹配时，用户已经从业务角度提出了专门的排名规则。在我们的例子中，老板希望列出的结果能够同时结合影片的受欢迎程度和发行时间。做到这一点需要同时采取两种排名手段，如图 7.9 所示。

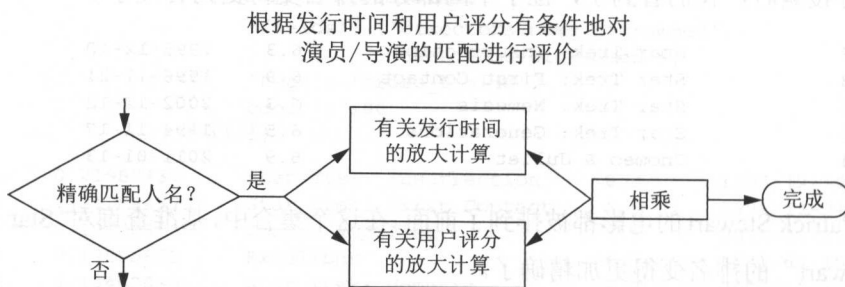


图 7.9 代表自定义业务排名规则的流程图，其规则针对的是演员或导演姓名的精确匹配。

首先，我们需要将那些高度准确的、针对人名的精确匹配放入它们自己的评价层中。我们知道，针对人名的精确匹配应该排在其他匹配的前面，因此它们应该被着重挑选出来。其次，也是本节提出的一个新观点，评价层需要拥有属于自己的排名方法。这样的方法需要建立在一定的基础之上，即与演员相关联的影片所具备的两个特征：影片发行的时效性和用户给影片的评分。

对于我们的查询，接下来这一部分，将包含三条规则：

- 针对人名的精确匹配本身，用以触发影片发行时间和用户评分这两个特征。
- 影片的实效性，基于其发行时间。
- 用户评分，介于 1~10 的一个平均得分，来自所有用户针对该影片的单独评分（1 代表最差，10 代表最好）。

我们在这一节里的工作，就是要将后两部分所产生的影响结合起来，但条件是仅当第一条规则（针对人名的精确匹配）被触发时才会生效。

在构造查询的这部分内容时，大家会逐渐开始意识到，这些查询的规模可以变得如此巨大！我们即将构造的，也许是大家曾经处理过的、规模最大的 Query DSL 查询。不过，我们正在学习处理这种庞然大物的办法，其方法和我们处理任何大规



模代码时所采用的思路是一样的。参与评价的每一个组成部分都有其目的性，都有一个参与整体排名函数的信号。当我们在工作中感到举步维艰时，那就毫不犹豫地处理问题的复杂度降到一个更加简单的层面上来：在将各个局部查询组合起来之前，先单独对它们进行处理。

我们如何将这一功能加入更大范围内的查询之中呢？这条查询会被纳入前一节中提到的布尔查询。我们最终将得到三个子句：前两个已经介绍过了，第三个是一条新的布尔查询。整条查询大致如下所示：

- SHOULD: 基准查询（参见清单 7.8 中的②）。
- SHOULD: 精确匹配片名的完整查询，放大 1000 倍（参见清单 7.8 中的①）。
- SHOULD: 包含人名全名的查询，放大 100 倍（参见清单 7.9）。
- SHOULD: 精确匹配人名的完整查询（本节新引入的）。

#### 7.4.5 忽略 $TF \times IDF$

我们想根据用户评分和发行时间为影片排名，但前提是仅当此查询为针对人名的精确匹配时。这样的排名根本不依赖于针对人名的精确文本查询，及其  $TF \times IDF$  值。这一点并不奇怪：坦白地说，许多时候我们需要的只不过是一个简单的 yes/no，而不是  $TF \times IDF$  值。在为 star trek 的片名进行放大处理时，我们早就已经注意到这一情况了。现在，让我们来深入探究一下针对人名的精确函数查询，向大家展示一下可以将  $TF \times IDF$  忽略掉的技术。

大家可能已经猜到了，因为针对该查询的评价将由两个数值类型的因素来决定，发行时间和用户评分，所以我们将采用 `function_score` 查询。回忆一下，使用函数查询需要指定一条带查询参数的基准查询，所以作为我们的初次尝试，不妨插入一条精确匹配人名的短语查询作为基准查询：

```
query = {
  "query": {
    "function_score": {
      "query": {
        "match_phrase": {
          "names_exact_match": SENTINEL_BEGIN + \
            " william shatner " + \
            SENTINEL_END
        }
      }
    }
  },
  ...
}
```

这里需要做一点改动；我们需要考虑到这样一个事实，针对该查询的  $TF \times IDF$  评价可能作用不大。要记住，默认情况下 `function_score` 查询会把针对查询的评价值与结果函数相乘。这意味着，我们会将精确匹配人名的  $TF \times IDF$  评价值与函数相乘。在某些场合下，我们需要一种方法，能够忽略掉  $TF \times IDF$  的评价值。为此，我们可以将查询放入 `constant_score` 查询内。这样做允许我们将结果查询（resulting query）的评价值硬编码为一个常量（即 `boost` 值）。由于我们关注的焦点是要把该评价值纳入一个具有更高区分度的层级，因此这里给它设置了一个非常高的值 1000：

```
"function_score": {
  "query": {
    "constant_score": {
      "query": {
        "match_phrase": {
          "names_exact_match": SENTINEL_BEGIN + \
                                " william shatner " + \
                                SENTINEL_END
        }
      },
      "boost": 1000.0}}
}
```

太棒了！现在，当用户的查询精确匹配演员或导演的姓名时，作为基准的相关性评价值就会是一个常量。

为了完成这一查询，我们还需要考虑到函数本身。这里有两个因素需要平衡：用户对影片的评分和影片的发行时间。每个因素都提供了一个可以单独进行调节的信号。所以，我们将在一定程度上对其进行单独处理。大家将会看到，各个信号是如何利用其自身的一系列技术，将其所产生的影响与用户期望保持一致的。保持这些信号的精确性并与用户的期望相一致，这一点是至关重要的。

## 7.4.6 捕捉综合质量指标

首先，让我们来看一看针对用户评分的放大处理。是时候回过头从数学的角度再仔细思考一下这个问题了。代表用户评分的字段（称为 `vote_average`）就是一个文档属性的例子，它可以直接对内容的价值进行度量。我们经常会对其他类似的、可以直接体现内容质量的度量指标进行放大，如：票房收益（profitability）、人气指数（popularity）、页面访问量（page views）等其他因素。在这一节中，伴随着对信号的反复打磨，我们将会深入探讨把这些综合质量特征包含进来的意义所在。我

们应该直接使用这些特征的取值还是对其进行一定程度的修改？也许应该降低其影响？或者应该放大其影响？用户对这一质量度量指标的重要性是如何看待的呢？

要结合使用 `vote_average`，就离不开 Elasticsearch 中的 `field_value_factor` 函数。该函数允许我们直接使用字段的取值。作为可选方案，我们也可以对字段做一些简单的修改，或代入函数计算。例如在清单 7.10 中，我们取 `vote_average` 的平方根，并将其乘以 2。

清单 7.10 将用户对影片的评分计算在内

```
query = {
  "query": {
    "function_score": {
      "query": {
        "match_all": {}
      },
      "functions": [
        {
          "field_value_factor": {
            "field": "vote_average",
            "modifier": "sqrt",
            "factor": 2
          }
        }
      ]
    }
  }
}
```

这一处理方式可以被可视化为如图 7.10 所示的曲线。

为什么要开平方呢？原因在于，我们寻找的信号要能够衡量用户对影片评分重要与否的态度。如果直接使用该取值，那么系统就会认为，一部评分为 10 分的影片，要比一部评分为 5 分的影片重要两倍。但用户实际上很少会认为质量因素所带来的影响会如此简单而直接。通过开平方，从图 7.10 所示的曲线可以看出，一部评分为 3 分的影片和一部评分为 10 分的影片相比，其重要性大约是后者的一半。

这些质量度量指标是很复杂的，我们很少直接将它们作为信号纳入运算。正如此处所做的那样，我们必须以包含这些特征的函数作为手段，搞清楚用户对函数取值的预期。例如，假设不用 1~10 这样的用户评分，而是用影片的 TMDb 主页浏览量来作为质量度量指标，那情况又会如何呢？大部分页面都只有个位数的浏览量，一小部分页面的浏览量达到了几百，少数达到了几千，而有一两个页面的浏览量则达到了几百万次。那么，一个达到了百万次浏览量的页面，在质量上就是其他页面的几百万倍吗？当然不是！对于我们的用户而言，也许以两倍的重要性来衡量会更合理一些。如何调低这些因素带来的影响是非常重要的。用户所考虑的因素，可能

要比我们简单粗暴地将质量度量指标直接纳入排名计算来得更加细致。

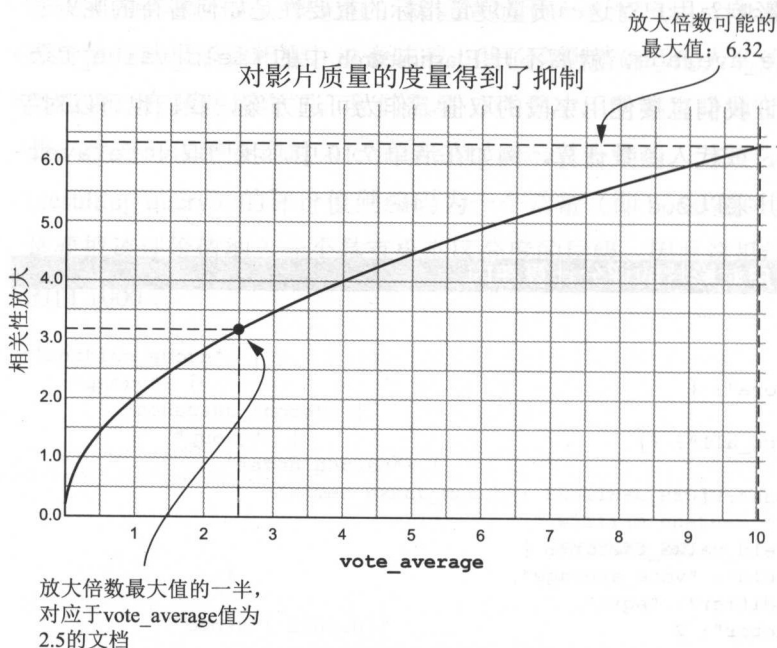


图 7.10 用户评分, 作为反映内容质量的指标, 得到了放大处理。此处采用开平方的方式减弱了其影响, 以便与用户对该度量值的理解保持一致。

### 7.4.7 达成用户的时效性目标

我们继续来制订精确匹配演员或导演姓名时所采用的自定义排名规则。本例中第二个需要考虑的信号, 是用户对影片发行时间在重要程度上的认知。本节中我们将会看到, 排名通常是对实际结果与用户所表述的目标在距离上的差距来进行建模的。例如, 用户想要对近期发生的新闻事件进行排名, 或者对住所附近的餐厅, 接近心理价位的电视机进行排名。Elasticsearch 内置的衰减函数 (decay functions) 就是根据文档和目标的距离远近来进行排名的。在本节中, 大家将会看到, 我们是如何根据 TMDB 的影片发行时间, 对一个简单的以用户目标为导向的排名问题进行建模的。

不过, 实现衰减函数的核心在于, 如何通过三个主要变量来告诉搜索引擎, 我们所要实现的目标是什么。这三个变量分别为: origin、decay 和 scale。origin 意指用户的目标 (或想法)。在 origin 所处的位置点, 文档的评价值是最高的 (为 1.0)。

scale 也是一个点，只不过处于较远的位置。在这一位置上，我们可以想象用户会这么说，“从目标出发到这一点，我觉得此时的文档价值已经很小了”。这就好像是用户在说，“如果为了吃一顿饭，我需要从家里出发开 20 分钟的车，那么我对这顿饭的兴趣也就只剩下一半了”。我们如何定义价值减少的程度呢？这就是 decay 参数的职责。给定 scale，decay 表明了在某位置点上的价值。将上述概念结合起来，我们可以重新修改一下用户对餐厅的表述：“如果为了吃一顿饭，我需要从 <ORIGIN> 出发开 <SCALE> 分钟的车，那么我对这顿饭的兴趣也就只剩下 <DECAY> 了”。

对于我们的查询，我们规定了发行时间早于 900 天（由 Elasticsearch 通过“same-date”语法帮助实现）的影片将只有大约 0.5 的评价值，或者被认为对用户的价值减半。我们将以此作为开始，用单独的查询表示如下，参见清单 7.11。

清单 7.11 距离用户时效性目标的高斯衰减

```
query = {
  "query": {
    "function_score": {
      "query": {
        "match_all": {}
      },
      "functions": [
        {
          "gauss": {
            "release_date": {
              "origin": "now",
              "scale": "900d",
              "decay": 0.5
            }
          }
        }
      ]
    }
  }
}
```

有关高斯衰减（Gaussian decay）的方程式比较复杂，但是为了能够让大家对其表现形式有所了解，我们在图 7.11 中给出了它的图形曲线。该曲线通过一个与距离当前天数相关的函数，反映了用户对涉及某位演员或导演的影片在价值层面的认知程度，它表明 900 天以前的影片只有一半的价值。而对于五六年前的电影，该函数所产生的影响则开始接近 0——变得几乎毫无价值了！

在此，大家可能会问，decay 函数是否有点太过激进了。也许 scale 设得太近了一些。为什么这么说呢？来看一个有可能存在的用例：一位正在搜索演员的用户想要的不仅是近期发行的影片而且还包括该名演员参演的经典影片。有些用户想要的是近期发行的影片，而有些用户想要的则是真正被奉为经典的影片，不论其发行时

间远近，我们需要仔细地在这两个彼此冲突的目标之间取得权衡。

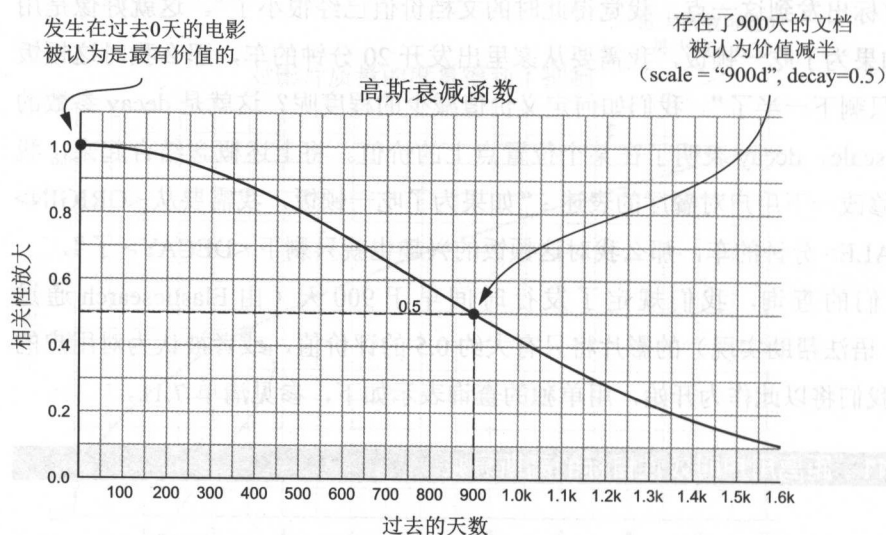


图 7.11 一个乘法放大，控制着距离用户目标的远近所产生的影响；此处的目标是：影片发行日期为“当前”，900 天之前发行的电影则被认为价值减半。

这是一个值得注意的地方，因为并非所有用户都是一模一样的。正如我们在后续章节中要讨论的那样，处理这一情形的一种选择可能是搞清楚何种用户具有何种优先级。然而，本章的目的是要我们提出一个合理的通用方案。前面所提议的函数认为，五六年前的电影就几乎分文不值，价值接近于 0 了。我们不太可能帮普通用户来做这样的决定，发行时间没有那么久的影片就一定是有价值的。目前阶段，让我们暂且将这个函数的衰减时间延长一些。不是 900 天，而是更保守一点的值：15 年。

图 7.12 中的曲线展示了修改后的函数。 $x$  轴以“千日”为单位。衰减过程是在几十年的时间里逐渐发生的，而不是短短的几年时间就衰减殆尽了。

我们所给出的这一函数，在匹配演员 / 导演的姓名时，对于刻画一般用户的时效性目标而言，很可能是一个更好的模型。目前我们的模型认为，发行超过 15 年的影片对用户来说价值减半。与任何其他信号一样，这只是一个起点。我们可以反复考察这一信号，评估其是否依然太过激进。（很多精彩的影片都出自 15 年之前！）通过对这些“旋钮”的微调来得到正确的信号，就是我们在不断地调整对用户目标的理解。幸运的是，我们现在已经掌握了针对用户目标进行相关性矫正所需要的所有工具。



价值最高的位置依然对应于过去0天

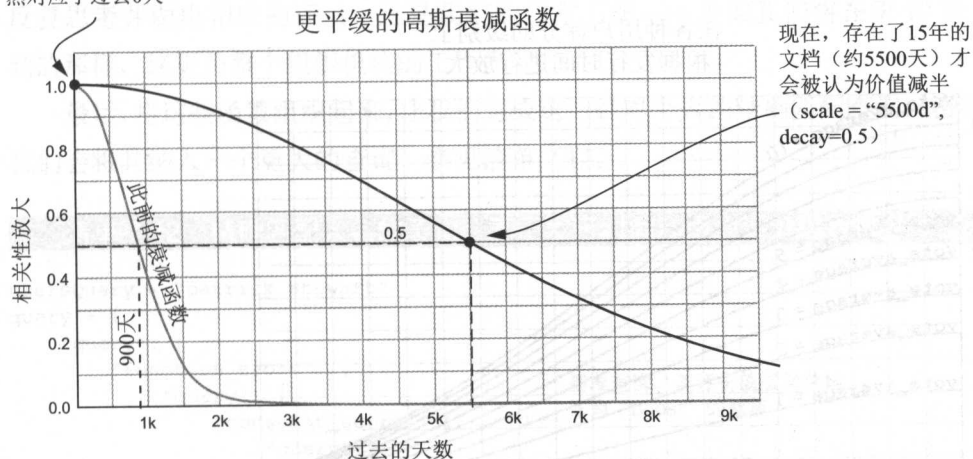


图 7.12 调整后的针对影片的乘法放大；15 年前发行的影片被认为价值变成了现在的一半。

这些衰减函数使我们能够根据考察对象距离用户目标的远近来产生信号。这是一种通用的模式，不只局限于日期时间和地理位置。来看一个真实的房地产搜索应用，其排名完全取决于房屋是否与用户目标相符。各种因素，诸如：周边学校质量、通勤距离远近、价格区间范围以及到临近公园的距离等，根据考察项目距离用户理想目标的远近，都可以用来参与排名。有时，这些排名问题并不被视为搜索问题，而更像是一种机器学习领域的问题，这里的界限已经开始变得比较模糊了。我们看到了，搜索引擎作为一个框架可以通过编程实现各种排名功能，其所依赖的不仅限于文本，还包括很多其他因素。

### 7.4.8 结合函数查询

是时候开始重新构建我们的主查询了。回想一下，我们一路走来，对于精确搜索导演姓名/演员姓名的用户，尝试为其建立了两个重要的信号。我们考虑到了电影评分和发行时间对用户而言的重要程度。现在重新开始对主排名函数进行编程。为此，我们来看一下，将前述两种曲线合并之后会有什么影响。

我们需要对两个变量相乘以后给相关性评价带来的影响进行可视化，以此来判断这种影响是否合适。可视化的一种方法是画一个非常酷的三维图。不过，有时将几个二维图重叠在一起，让每个二维图代表一个变量的变化，对于一些关键性问题的解答可能是一种更加有效的方法。图 7.13 中的曲线就代表了影片的评分函数与衰

减函数相乘之后得到的结果。

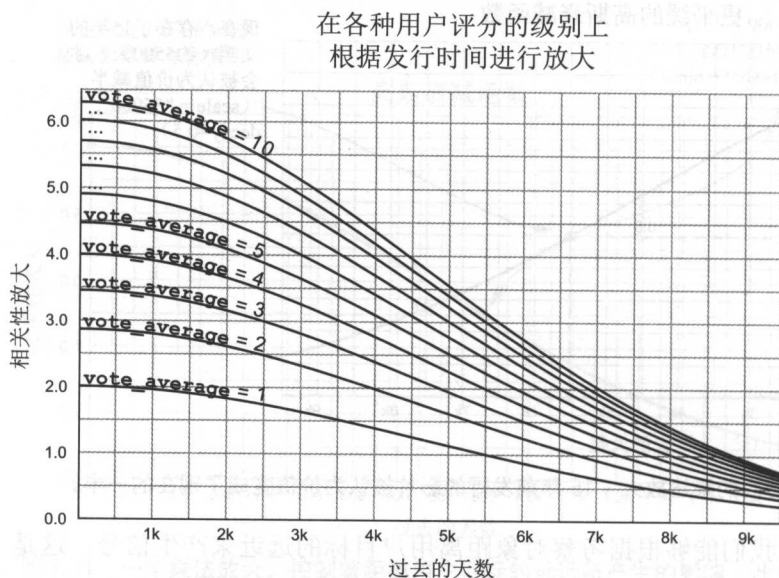


图 7.13 基于用户对某部影片的评分字段 (vote\_average) 得到的衰减曲线，演示了发行时间和用户评分对于精确匹配人名的相关性评价所构成的共同影响。

最下面那条线对应于影片衰减曲线图中评分为 1 的曲线。最上面的那条线则对应于用户评分为 10 的影片。关于发行时间和用户评分这两个因素的组合，这张图告诉了我们什么样的信息呢？有一点我们需要注意，那就是评分为 1 的影片与评分为 10 的影片之间的关联关系。从现在算起，在过去的哪个时间点上，对于评分为 10 的影片，其相关性评价就开始小于评分为 1 的影片了呢？这一刻发生在 7000<sup>1</sup> 天再往前一点。

我们可以思考一下前述这两个因素结合之后所产生的影响。它们是如何共同作用得到信号的相关性结果的呢？在本例中，这些函数是代表相关性计算的，这一点毫无疑问。对每一个函数协同工作的机制单独进行考察有助于我们分析其所产生的影响。这两个因素都具备了正确的优先级吗？还有什么需要改善的地方吗？或许发行时间应该有一个年限；例如，可能用户更关注最近发行的电影，19 世纪 80 年代

<sup>1</sup> 此处原书中为 6000 天，我认为原作者可能存在笔误。根据我的理解，此处应该是和 vote\_average=1 且时效为 0 的相关性放大值进行比较。因此就是从  $y=2.0$  出发画一条横线，会在大约  $x=7k$  左右的位置与 vote\_average=10 的曲线相交。也就是说，对于用户评分为 10 的影片，当随时间衰减到 7000 天之后，其评价结果就会低于当前发行的用户评价仅为 1 的影片。——译者注

的电影和 19 世纪 70 年代的电影对他们来说没有多大的区别。对所有这些因素进行反复思考并做出相应的评估，这是我们的日常工作——正如我们将在第 10 章中看到的那样，这更是整个组织机构的工作内容！

最后，将这些函数组织起来，让我们完成这一针对个人的精确匹配查询。相应的，我们会将其纳入一个更大的查询，参见清单 7.12。

清单 7.12 精确匹配人名的子句，基于发行时间和用户评分进行排名

```
usersQuery = "patrick stewart"
query = {
  "query": {
    "function_score": {
      "query": {
        "constant_score": {
          "query": {
            "match_phrase": {
              "names_exact_match": SENTINEL_BEGIN + " " + \
                                  usersSearch + " " + \
                                  SENTINEL_END
            }
          },
          "boost": 1000.0
        },
        "functions": [
          {
            "gauss": {
              "release_date": {
                "origin": "now",
                "scale": "5500d",
                "decay": 0.5
              }
            }
          },
          {
            "field_value_factor": {
              "field": "vote_average",
              "modifier": "sqrt"
            }
          }
        ]
      }
    }
  }
}
search(query)
```

基准查询，其评价价值被简单地作为“boost”的取值

针对时效的放大

针对评分的放大

Results:

Num	Relevance Score	Movie Title	vote_average	release_date
1	2.762838	X-Men: Days of Future Past	7.7	2014-05-23
2	2.4984634	The Wolverine	6.4	2013-07-25
3	2.4377568	Ted	6.3	2012-06-29
4	2.2800508	Gnomeo & Juliet	5.9	2011-01-13
5	1.9779315	TMNT	6.0	2007-03-22

值得注意的是，我们针对“Patrick Stewart”的搜索，得到的结果中排在第一位的，是一部近期发行的、用户评分相对较高的影片。同样要注意的是，排名第 4 位

和第5位的影片之间的差别。一部评分为5.9、发行于2011年的影片，其排名要先于一部评分为6.0、发行于2007年的影片。看起来这样的预期似乎和前面图中所展现的不同评分带来的影响是相符合的。

### 7.4.9 把一切联系起来

上述查询只不过是一个更大范围内的查询所包含的一个子句而已！不过这里我们就不列出完整的查询了（大家可以访问 GitHub 上的例子，一窥完整查询的全部细节）。尽管如此，概括地说，整个查询可以分为4个布尔子句，包括如下规则：

- 整个查询应该（SHOULD）精确匹配片名（参见清单7.8中的①）。
- 整个查询应该（SHOULD）精确匹配导演或演员的全名，并且是按热门程度和发行时间进行评价的（参见清单7.12）。
- 查询中有一部分应该（SHOULD）匹配导演或演员的全名（参见清单7.9）。
- 用户的所有查询词应该（SHOULD）在文档中有所匹配（参见清单7.8中的②）。

将这4个 SHOULD 子句组合起来放入一个布尔查询，我们就大功告成了！我们解决了一个极其复杂的排名问题：我们用英语表达了自己的想法以及需求等，并将其放入搜索引擎——让搜索引擎能够如我们所愿！

现在我们可以开始问更大一点的问题了。和我们在实际编程中一样，一旦知道了自己该如何编程，可能会突然意识到一个更大的问题：我们究竟应该针对什么进行编程呢？同样的，随着对相关性编程越来越熟练，我们会逐渐意识到相关性处理中也存在这样的挑战。随着我们对后续章节的学习，大家会发现，找到正确的排名需求将会成为重中之重。如何定义，以及定义什么样的需求/用例，对搜索而言才是重要的呢？如何通过测试，确保我们依旧满足用户的需求呢？此处，我们将这些问题留给大家作为一项技术上的挑战。在后续的章节中，我们将会看到，现实生活从来就没那么简单！

## 7.5 本章小结

- 评价调整（score shaping），包括像放大（boosting）和过滤（filtering）之类的技术，是对搜索结果的排名进行编程，以满足商业/用户需求。
- 放大处理（boosting）包括若干种形式，如，布尔查询与函数查询，加法处

理与乘法处理。

- 布尔查询对排名做了抽象，它提供一种简单的方式可以增加放大倍数。对其进行校准就是正确地将加法放大的放大倍数一层层叠加上去。
- 函数查询允许我们通过任意形式的放大处理来控制排名算法。对其进行校准意味着从数学角度对用户的优先级进行建模。
- 过滤器提供的通常是放大处理的一种替代方案，它不是要试图提升高优先级结果的排名，而是要剔除低优先级的结果。
- 评价调整的效果取决于我们是否能实现高质量的信号，并且能根据业务规则将其纳入排名函数。
- 对于高质量的信号，我们可以将它们放入自己的评价层，以提升其优先级。
- 我们可以采用各种广泛的排名方式，包括对用户目标进行建模、结合内容质量的度量指标等。

# 8 提供相关性反馈

## 本章要点

- 用户与搜索应用之间持续不断地进行多方面对话
- 除了相关性排名，能带给用户相关性内容的方法还包括：
  - 引导用户选择更好的搜索查询
  - 通过拼写检查纠正用户的搜索错误
  - 突出显示文档之所以对用户而言具有相关性的原因
  - 向用户解释搜索的处理过程
  - 允许用户从结果集中过滤掉无关的内容

截至目前，我们已经讨论过对相关性排名进行深入控制的方法。在本章中，我们将会看到，相关性排名并不是唯一能给用户带来相关性内容的方法。搜索为我们提供了介于用户和搜索引擎之间的多层面的交互手段。由于这种交互在相关性方面的表现从来都不会是完美的，因此用户对于我们提供额外的手段为其带来相关性内容，总是会心存感激的。在本章中，我们将通过开发如下功能，来驾驭除相关性排



名之外的，从多个层面与搜索引擎进行交互的方法：

- 向用户解释查询的处理过程。
- 纠正用户在输入和拼写上的错误。
- 向用户推荐其他搜索，以带来更好的搜索结果。
- 向用户揭示系统所理解的文档在集合中的分布规律。
- 帮助用户理解某篇文档与查询相匹配的原因。
- 帮助用户有效地理解结果集。

我们称这种交互式的辅助手段为相关性反馈（relevance feedback）。在本章，我们会给出相关性反馈的概述，内容涵盖搜索用户体验的三个层面：搜索框（search box）、浏览与过滤（browsing and filtering），以及搜索结果（search results）。图 8.1 展示了将在本章中讨论的这几项活动；相关性技术研发工程师的工作就是要实现能够支持这些活动的功能，在搜索应用的各个不同阶段引导用户获得相关性内容。

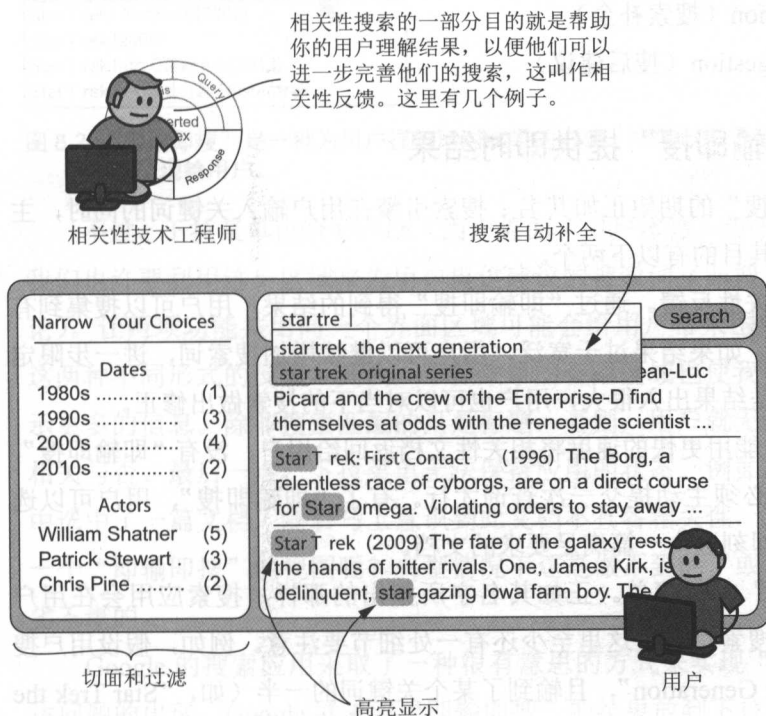


图 8.1 相关性反馈帮助用户细化搜索并找到所需的信息

在这项工作中，我们会发现，与获得完美的相关性排名相比，引导用户的搜

索行为可能是一个更容易解决的问题。在本章中，我们将为用户获得相关性内容另辟蹊径。相关性反馈的形式不计其数，实现它们的方法也不胜枚举，对此我们不会一一列举。相反，我们希望为大家开启一条成就相关性技术工程师的重要途径：利用一切可能的办法，实现用户与搜索之间的双向交互。

## 8.1 搜索框中的相关性反馈

用户和搜索应用的交互首先是从搜索框开始的。对于利用搜索框进行交互究竟能带给用户多少相关性反馈，也许你会感到惊讶。通过在用户输入搜索词时，以及提交查询之后，为其提供即时的辅助信息，我们让用户得以对查询进行细化，甚至有的时候在提交查询之前就可以找到他们所要的结果。本节我们将围绕搜索框讨论三种常见形式的相关性反馈：

- search-as-you-type（即输即搜）
- search completion（搜索补全）
- postsearch suggestion（搜后建议）

### 8.1.1 利用“即输即搜”提供即时结果

我们对“即输即搜”的期望正如其名：搜索引擎在用户输入关键词的同时，主动提供匹配的文档。其目的有以下两个。

第一个目的是相关性反馈。通过“即输即搜”得到的结果，用户可以搜集到有关查询有效性的信息。如果结果过于宽泛，用户可以继续增加搜索词，进一步限定其所寻找的文档。要是结果出入很大，用户也可以对当下的搜索做出修正。

其次，这样做也许能用更快的速度将相关性文档返回给用户。没有“即输即搜”，用户在得到结果之前必须主动提交一次查询才行。有了“即输即搜”，用户可以选择退出当前输入，或即刻选中一篇满足要求的文档。

实现“即输即搜”非常简单。正如其名字所指出的那样，搜索应用会在用户输入期间发起一系列搜索。不过这里至少还有一处细节要注意。例如，假设用户搜索“Star Trek the Next Generation”，且输到了某个关键词的一半（如，“Star Trek the Next Gener”），此时我们不应该将尾部的这半个单词包含在搜索中，且没有指名它是一个前缀，而非完整单词。所幸的是，Elasticsearch 提供了一种 `match_phrase_`

prefix 查询，实现了这一思路。看看下面这条查询：

```
{ "query": {  
  "match_phrase_prefix" : {  
    "title" : "star trek the next gener"}}}
```

如果将该查询交给 `/_validate/query?explain` 服务，其解释信息所呈现的情况和我们预计的一样，此查询被解释成了一条短语查询（phrase query），最后一个单词凭借其末尾的通配符得到了扩展：

```
"explanation": "title:\"star trek the next gener*\""
```

如图 8.2 所示，典型情况下，应用会把“即输即搜”的匹配结果直接显示在一个下拉菜单中。该菜单使用户得以轻松地选择匹配的文档。

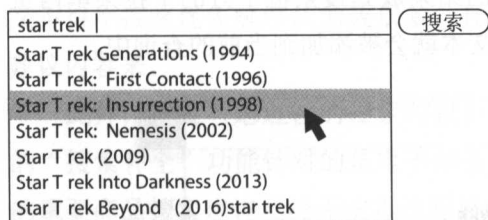


图 8.2 “即输即搜”是一种为用户提供即时结果的方法。它常以位于搜索框下方的下拉菜单的形式呈现给用户。

不过，下拉菜单也给 UI/UX（用户界面 / 用户体验）带来了一定的挑战。首先，我们也许要利用这一区域来为用户提供建议的搜索词（即搜索补全，将在下一节讨论）。让两项功能共用同一个界面区域可能会给用户带来困惑，他们必须能够区分这两种不同形式的反馈。其次，下拉菜单的有限区域也使我们很难为匹配的文档显示更多的信息。除非搜索匹配的只是标题，否则，用户就无法总能成功判断出文档相关与否。最后一点，下拉菜单无法保持应用的状态。例如，假设用户从下拉菜单中选中了一篇文档，随后马上意识到此文档不具备相关性，用户如何才能切换回上一个“即输即搜”的视图呢？典型情况下这是做不到的，或者即使能做到也多是拙劣不堪的。

Google 的搜索应用采取了一种很有意思的方式来实现“即输即搜”，避免了上述问题的出现。Google 并未将“即输即搜”的结果放到下拉菜单里，而是将其放入了通常用于显示搜索结果的区域内。这种处理方式有助于避免用户可能出现的困惑。将结果置于其通常应出现的地方，而保留下拉菜单用于“搜索补全”给出的建议信息。

如此一来，用户就能做到要么将焦点放在搜索结果上，要么放在“搜索补全”的建议上。而将搜索结果置于其通常应该出现的地方，这种做法相比于下拉菜单，也有了更多的空间，可以让我们为搜索结果提供更为丰富的信息。

### 8.1.2 利用“搜索补全”帮助用户找到最佳查询

搜索补全可以引导用户获得更优质的搜索查询和关键字。事实上，搜索应用是就用户所查找的内容在与用户一起进行头脑风暴：“用户，您输入了 robots are，您是想搜 robots are our friends 吗？或者是 robots are awesome？”如果我们对这一交互过程进行持续微调，伴随用户不断输入关键字，反复设定搜索词，并间或从自动补全的匹配中选择若干结果，整个过程就几乎变成了一种下意识的行为。

如图 8.3 所示，搜索应用通常会把补全的结果放到搜索框下方的下拉菜单内呈现。选择下拉条目中的任何一项，所选中的文本就会被添加到当前的查询中。

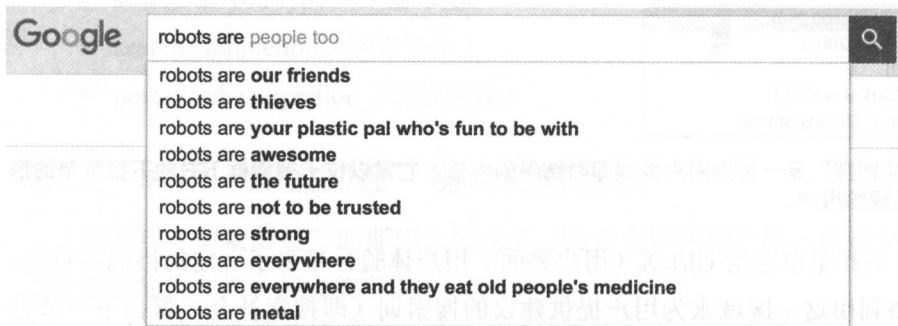


图 8.3 通常，“搜索补全”会被放到搜索框下方的下拉菜单内。从下拉结果中选择一项，用选中的文本替换当前查询。

要实现“搜索补全”可能是有挑战的。用户希望“搜索补全”在他们开始输入之后马上就能返回结果。如果速度太慢，用户就会觉得补全功能有问题。于是用户就得不到我们的引导，只能在没有提供辅助信息的情况下独自完成搜索了。补全的结果还必须是高度相关的。否则用户会忽略不相关的建议，并转而继续在没有引导的情况下操作。类似的，如果用户选择了一条补全建议，但是得到的结果出乎他的意料，或者匹配不到任何结果，用户也会觉得被误导。

和大多数本章所讨论的功能一样，我们可以用许多方法来实现搜索补全。在接下来的讨论中，我们将给出三种方法。

### 根据用户输入产生补全

当我们在准备生成搜索补全时，首先必须要问自己一个问题：“应该从什么样的数据源中提取补全文本？”在拥有足够使用量的情况下，我们可以依据用户过往的查询来得到补全建议。这样做是有道理的：允许用户依据其他用户过往的搜索情况来进行搜索，应当可以确保当前用户不会偏离常规情况太远，不是吗？

也许的确如此——但也要小心；我们需要考虑几个问题。首先，让我们来看看拥有足够使用量这一假设。我们的搜索应用是一种短尾（short-tail）应用，只要少量查询（也许数百条）就可以代表搜索使用量的主体吗？抑或是一种长尾（long-tail）应用，数千条查询也只代表了一小部分搜索使用量？如果查询数量太少，也许我们就不会有足够的数据量来获得令用户满意的补全体验。而如果查询数量过多，我们又不得不按照优先顺序从为数众多，且花样百出的候选补全结果中选出最为重要的建议内容来。

此外，还要考虑过往的搜索对我们的应用而言是否已经过时的情况。假设我们的“搜索补全”功能针对的是电子商务应用，其库存经常发生变化。那么一个月前的重要商品搜索也许已经无法再匹配目前可供选购的商品了。

最后一点，可能也是最为麻烦的，过往的用户查询是一个源自文档集的离散数据集。我们很容易找到匹配结果为 0 的用户查询！将这样的查询作为“搜索补全”的结果推荐给用户，会让用户感到迷惑不解。如果“搜索补全”带给用户的体验很差，那么用户就不会再信任我们的搜索应用了。

基于用户过去查询结果的“搜索补全”可以提供良好的用户体验；但作为搜索研发工程师，我们必须确保为用户的查询提供足够丰富和及时的补全信息。此外，我们也有理由确保搜索用户得到的补全信息，应该是有助于获得相关性搜索结果的。

### 根据被搜索文档产生补全

生成“搜索补全”的一种更为直接的方法是利用被搜索内容的文本。该方法确保了任何推荐给用户的补全建议都是与文档集中出现的文字紧密相关的。

让我们来看一看，利用 TMDB 数据集中的文本，我们可以如何实现“搜索补全”。在这一过程中，我们首先需要考虑的是哪些字段可以被用来作为补全的依据。针对 TMDB，假设我们只考虑两个字段：title（片名）和 overview（简介）。有些用户以片名来搜索电影。对这些用户而言，很明显 title 字段是有效的。而另一些用

户则通过输入影片详情来进行搜索。对这些用户的“搜索补全”而言，更为丰富的 overview 字段提供了更好的素材。

不过丰富的文本也带来了副作用。因为有太多的文本，overview 字段具有很大的多样性和噪声。用户可能会对看似很随意地从 overview 文本中提取的补全建议感到失望。对于这些篇幅冗长的字段，我们得想想办法，对其按优先级进行排序，看看哪些搜索补全应该排在前面，这属于“搜索补全”自身的排名和相关性问题。不过眼下，还是让我们保持问题的简单化，只用 title 字段作为补全文本的来源吧。

接下来要考虑的问题是应该如何对文本进行分析。在第4章中，我们探讨了文本分析的几种方法，以帮助用户找到所需。在那一章中，我们使用了词干处理，以及其他形式的 token 处理手法，将诸如 happy、happier、happiest 这样的单词统一成一个 token：happi。这些 token 是不适合作为“搜索补全”的数据源的。用户输入单词的一部分“happ”时，系统会给出 happi 这样的补全建议，这压根就不是一个单词！

相反，我们要在分析过程中保留单词的可读性，如清单 8.1 所示。

#### 清单 8.1 对基于双联词的补全，分析其所需的配置

```
"settings": {
  "analysis": {
    "filter": {
      "shingle_2": {
        "type": "shingle",
        "output_unigrams": "false"
      }
    },
    "analyzer": {
      "completion_analyzer": {
        "tokenizer": "standard",
        "filter": [
          "standard",
          "lowercase",
          "shingle_2"
        ]
      }
    }
  }
}
```

① 针对双联词的 shingle 过滤器

用来生成补全文本的补全分析器

我们创建了一个名为 completion\_analyzer 的分析器。该分析器（利用标准分词器）按标点和空格对文本进行切分，并将 token 转换成小写。小写转换忽略了内容或搜索字符串的大小写，有助于内容的匹配及推荐。作为额外的一步，为了支持短语类型的建议，我们使用了 shingle 过滤器，用来生成由两个单词构成的短语。分析器会将字符串 star trek: into darkness 转换成若干个 token：star trek, trek into, into darkness。



既然有了 `completion_analyzer`，那就让我们将其用于 `title` 字段的文本处理吧。为此，我们需要将 `title` 字段复制到另一个名叫 `completion` 的字段，用 `completion_analyzer` 对其进行分析，如清单 8.2 所示。

清单 8.2 基于 `title` 补全的映射定义

```
"mappings": {
  "movie": {
    "properties": {
      "title": {
        "type": "string",
        "analyzer": "english",
        "copy_to": ["completion"]},
      "completion": {
        "type": "string",
        "analyzer": "completion_analyzer"}}}}}
```

将 `title` 字段复制到 `completion` 字段

`completion` 字段用来保存补全文本

最后，在完成对文档的索引之后，我们就可以构造带补全功能的搜索了。这样的搜索会产生出与用户输入查询相匹配、由两个单词构成的、最常见的短语，参见清单 8.3。

清单 8.3 用以生成自动补全的查询

```
{ "query": {
  "match_phrase_prefix": {
    "title": {
      "query": user_input}},
  "aggregations": {
    "completion": {
      "terms": {
        "field": "completion",
        "include": completion_prefix + ".*"}}}}}
```

① 限定候选的补全项为所有匹配用户查询的片名

② 推荐的搜索补全项来自 `title` 字段的文本

③ 限定推荐的补全项为以 `"completion_prefix"` 开头的词

让我们来看一看清单 8.3 的工作原理，看它是如何生成补全项的。请注意这两个变量，`user_input`① 和 `completion_prefix`③。

变量 `user_input` 保存了由用户输入的完整查询字串。如果用户输入了“Star Tr”，则 `user_input` 为 `star tr`。我们将 `user_input` 放入之前在介绍“即输即搜”时引入的 `match_phrase_prefix` 查询中。该查询强制补全项使用与用户查询相匹配的电影片名。在本例中，这样做可以避免我们使用像 `star wa movies` 这样的文本作为补全项推荐给“星际迷航”的粉丝们。

补全项的前缀③是用户当前输入的单词。对 Star Trek 而言, 即 `tr`。这样做就将补全项限定在了用户可能会尝试输入的短语范围内了(即以 `tr` 开头的短语)。有时候, 尤其是当最后一个输入的单词过短时, 这种做法能将前面的单词也包含进来, 从而有助于对补全项做进一步限定(即以 `star tr` 开头的短语)。如果连文本都还没有给足(比如用户刚刚开始输入), 那么也许我们会希望完全忽略补全项。因为此时为用户提供帮助的上下文信息太少了。

让我们再次参看清单 8.3, 此处我们利用了 Elasticsearch 的 `term` 聚合②来获取搜索完成项。给定一条查询(在本例中, 即 `match_phrase_prefix` 查询), `term` 聚合搜集的, 是所有在匹配该查询的文档中出现的单词所构成的列表。Elasticsearch 返回单词的列表, 根据包含单词的文档数对其进行排序。在③处, 我们告诉 Elasticsearch, 对单词加以限制, 只包含那些以 `completion_prefix(tr)` 打头的单词。

这种方法得到的是一组与输入查询相匹配的, 最为常见的 `title` 短语。向系统发起清单 8.3 中定义的搜索后得到的返回结果, 包含了一组补全项, 如清单 8.4 所示。

#### 清单 8.4 基于聚合方法的“搜索补全”得到的返回示例

```
{'completion': {'buckets': [
  {'doc_count': 1, 'key': 'trek 3'},
  {'doc_count': 1, 'key': 'trek axanar'},
  {'doc_count': 1, 'key': 'trek first'},
  {'doc_count': 1, 'key': 'trek generations'},
  {'doc_count': 1, 'key': 'trek horizon'},
  {'doc_count': 1, 'key': 'trek ii'},
  {'doc_count': 1, 'key': 'trek iii'},
  {'doc_count': 1, 'key': 'trek insurrection'},
  {'doc_count': 1, 'key': 'trek into'},
  {'doc_count': 1, 'key': 'trek iv'}
], 'doc_count_error_upper_bound': 0, 'sum_other_doc_count': 4}}
```

利用聚合方法生成的补全项, 确保了补全项在用户已有输入的上下文中与我们的文档相匹配。例如, 假设用户输入了部分查询“Star Tre”, 则在本例中, 补全项前缀为 `tre*`。基于该前缀, 我们得到的补全项有可能会是诸如 `treasure island` 和 `treading water` 这样的短语, 但没有一个是和 `star` 相关的。由于聚合考虑到了搜索的上下文, 因此针对 `star tre` 所返回的补全项, 只会包含适合于当前上下文(context-appropriate)的内容, 比如 `trek generations` 和 `trek insurrection`。表 8.1 列出了一些补全项的例子, 可以被当作用户搜索, 用来查询“星际迷航”的影片。

表 8.1 “针对“星际迷航”影片，可用于用户搜索的补全项

用户输入	补全项前缀	补全项
st	(未采用——上下文信息太少)	(不适用)
star	star	star wars star trek starship troopers
star t	star t	star trek
star tr	tr	trek 3 trek axanar trek first trek generations

上述方法的另一个附带的好处是，它可以很容易地与“即输即搜”结合使用，因为正如我们在清单 8.3 中所见的那样，我们的确是在用户通过 `match_phrase_prefix` 输入关键词时发起的搜索请求<sup>①</sup>。

毫无疑问，我们可以对上述方法进行改进。当前的策略是将补全项限定在两个单词以内。要是补全项能提供整部电影的片名，那就更好了。利用能生成更长补全项的其他分词策略，我们是有可能做到这一点的。例如，我们可以利用 `path_hierarchy` 分词器，将 `delimiter` 设为空格，并将 `reverse` 设为 `true`，对片名进行分词。这样做会将电影片名以空格进行切分，并且会将片名的结尾部分作为 `token` 保存起来。例如，*Star Trek: The Motion Picture* 会被切分成 `star trek the motion picture`、`trek the motion picture`、`the motion picture`、`motion picture` 以及 `picture`。这些 `token` 看上去做补全项再合适不过了！

使用上述方法有一点需要注意：和较长的 `token` 相比，只由一个单词构成的 `token` 出现的几率会变得更加多。但是，通过将 `term` 聚合所返回的补全项以普遍性从低到高的顺序倒序排列，我们就能为用户提供最具针对性，同时也通常是长度最大的补全项了。

在开始实现基于聚合的“搜索补全”之前，我们需要了解该方法存在的缺陷。首先，对大数据量的文本字段进行繁重的聚合处理，可能会给搜索引擎带来沉重的负担，尤其是在采用分布式搜索的应用中。如果我们选择使用该方法，请确保请求响应时间的延迟是可接受的；否则，我们给出的补全项就会赶不上用户按键的速度。因为这一原因，该方法在文档集不大，且文本字段所含的不重复词汇（`unique terms`）的数量相对较小的情况下最为有效。

其次，默认情况下该方法所返回的补全建议，是按照出现频率从最常出现到最不常出现这样的顺序来进行排序的。这样的衡量标准并非总是合理的。例如，参看清单 8.4，请注意所有可能的补全项在索引中都只出现了一次。因此，在该例中，想基于补全项在索引中出现频率的多少来对其进行排序是不可能的。为了解决这些问题，我们引入最后一种“搜索补全”策略：补全项专用索引（specialized completion indexes）。

### 借助专用索引快速产生搜索补全项

由于补全项是相关性反馈中很重要的元素，因此 Elasticsearch 专门引入了一种部件叫作 completion suggester。它克服了前面提到的性能问题，并允许用户自定义补全结果的排序规则（而非总是以出现次数为序）。事实上，completion suggester 是一种特殊的索引，它与普通索引一起同时被存储起来。支撑其工作的是一种压缩数据结构（一种有穷状态转换器，即 finite state transducer），该结构提供了一种快速前缀查找（fast prefix-lookup）能力。在许多方面，该方法都是解决补全项问题的最佳方案，但是正如我们稍后会看到的那样，它自身也带来了若干问题。

配置 completion suggester 很简单：我们只要将某个字段声明为 completion 类型即可，如清单 8.5 所示。

清单 8.5 配置 Elasticsearch 的 completion suggester

```
{ "mappings": {  
  "movie": {  
    "properties": {  
      "title": {  
        "type": "string",  
        "analyzer": "english"},  
      "completion": {  
        "type": "completion"}}}}}
```

completion类型的字段使用Elasticsearch的 completion suggester

原则上，我们可以将 title 字段复制到 completion 字段，并以前一节中演示过的同样的方式对文档进行索引。但是如果我們这样做的话，就失去了使用 completion suggester 的一个最大的好处：直接为补全项指定权重的能力。该权重影响着我們推荐补全项的顺序。在清单 8.6 中，我们为给定文档增加了一个新的 completion 字段，该字段用电影片名作为补全项的文本，并用影片受欢迎的程度作为补全项的权重。

**清单 8.6 为文档增加一个 completion 字段，将受欢迎程度作为权重**

```
doc = {  
  "title": "Star Trek Into Darkness",  
  "popularity": 32.15,  
  /*...other fields*/ }  
  
doc["completion"] = {  
  "input": [doc["title"]],  
  "weight": int(doc["popularity"])}  
}
```

原始的文档

对文档进行充实

权重必须是整数

在给这些得到充实的文档建好索引以后，我们就可以执行带补全项的查询了。在清单 8.7 中，我们采用的是 Elasticsearch 的 `_suggest` 服务，而非之前用过的 `_search` 服务。（不过假如你愿意的话，也可以在普通的 Elasticsearch 搜索中包含 `suggest` 子句，并从搜索结果中获得搜索建议。）

**清单 8.7 通过 `_suggest` 服务获得搜索补全项**

```
GET /tmdb/_suggest  
{ "title_completion": {  
  "text": "star tr",  
  "completion": { "field": "completion" } } }
```

在本例中，搜索的补全项以 `star tr` 文本为前缀。和之前的例子一样，我们得到的结果是一组“星际迷航”的影片，但这次是按影片的受欢迎程度而非出现次数来排序的。这样做给用户带来了更具相关性的搜索结果。此外，我们还可以将 `completion suggester` 配置成执行模糊匹配，从而可以让系统返回合适的补全项，而不必担心用户存在某种程度上的输入错误。

不过 `completion suggester` 再好也是有代价的。因为 Elasticsearch 是将 `completion suggester` 作为一种单独的索引来实现的，它并不了解搜索的全部上下文。例如，假设一位用户想寻找有关 Spock 殒命的“星际迷航”电影（抱歉，要剧透了）。如用户搜的是“Spock Dies Star Trek”会出现什么情况呢？很显然，没有哪个有关片名的补全项会以 `Spock dies star` 开头，因而一个好的补全项实现方案会试图寻找以第二个词打头的补全项——`dies star`——同样，还是找不到补全项。随后该实现方案会继续寻找只以 `star` 打头的补全项。此处，基于整个查询的上下文，返回除“星际迷航”影片以外的任何其他内容都是毫无道理的。但就 Elasticsearch 的 `completion suggester` 而言，排在最前面的补全项是 *Star Wars: Episode IV—A New*

*Hope*。这个例子表明，completion suggester 对搜索上下文是一无所知的。

使用有穷状态转换器获得补全项带来的另一个麻烦是，其数据结构是无法更改的。如果一篇文档从索引中被删除，针对该文档的补全项依然会存在。目前来说，解决这一问题的唯一办法是进行一次完整的索引优化（full-index optimization），实际上就是为补全项重新建一次索引。

抛开可能的缺陷不说，completion suggester 是一种提供相关性反馈的重要工具。Elasticsearch 的 completion suggester 偶尔会将用户带入歧途，但是，它允许我们定义规则，指定补全项以何种顺序进行排序（在本例中，即影片的受欢迎程度）。同时，典型情况下，completion suggester 还是获取补全建议项最快的一种方法。这种针对用户的快速反馈可以变成一种接近下意识的辅助手段，帮助引导用户选择更有针对性的搜索，并最终得到他们所期望的文档。

### 哪一种“搜索补全”的方法最好

在前面几节里，我们讨论了三种“搜索补全”的方法：根据用户输入进行补全，根据存在于索引中的文本进行补全，以及根据专门的索引进行补全。每一项技术都有助于令我们的用户更加接近其所寻找的信息。但正如我们已经看到的那样，没有什么所谓的银弹；每一项技术都有其优缺点。具备了这些知识，至少我们会有一个更好的起点，来构建我们自己的补全解决方案。

#### 8.1.3 利用搜索建议来修正输入和拼写错误

“搜索补全”在用户输入时为其提供建议。但当查询被提交之后，为用户提供相关性反馈的另一个机会又来了。用户在输入搜索时偶有犯错——拼写错误或输入错误。当他们提交了错误的查询之后，搜索引擎应能给出有关该查询的修改和改进建议。

此处，Google 再一次成为典范，它向我们示范了“搜后建议”是如何为用户提供相关性反馈的。如图 8.4 所示，假如 Google 收到了一条明确包含输入错误的查询，它会将查询内容替换成用户可能想搜的正确内容。类似的，假如用户查询存在输入错误的可能性，但情况相比而言还不甚清晰，则 Google 不仅会给出与用户原始搜索相对应的查询结果，还会建议用户选择一条可能更符合其目标的不一样的查询。无论上述哪种情况，此类信息的展现方式都是至关重要的。如果用户不知道自己的查询被替换过，也许他们就会被搞得云里雾里，开始对搜索应用产生怀疑。相反，如果系统（替换了查询却）从来不曾告诉用户其有输入错误，他们同样也可能会感到



失望，认为搜索应用无法找出其所期望的结果。

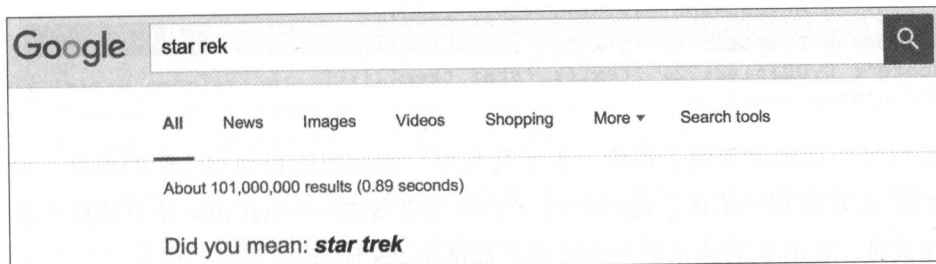


图 8.4 Google 利用“搜后建议”为用户提供相关性反馈

我们可以利用 Elasticsearch 的 phrase suggester 来实现“搜后建议”。为了对 phrase suggester 进行设置，其映射必须包含一个专门用于向用户提供建议的字段。与之前讨论过的补全项情况类似，我们必须对该字段进行分词，这样 token 才能依旧保持可读性，而且是拼写正确的单词。对于本例，我们会将 title 字段的内容复制到 suggestion 字段，并使用默认的标准分析链：

```
"mappings": {
  "movie": {
    "properties": {
      "genres": {
        "properties": {
          "name": {
            "type": "string",
            "index": "not_analyzed"
          }
        }
      },
      "title": {
        "type": "string",
        "analyzer": "english",
        "copy_to": ["suggestion"],
        "suggestion": {
          "type": "string"
        }
      }
    }
  }
}
```

为了获得建议信息，我们再次使用 \_suggest 服务：

```
GET /tmdb/_suggest
{ "title_suggestion": {
  "text": "star trec",
  "phrase": {
    "field": "suggestion"
  }
}
```

上述调用返回的建议项如下所示：

```
{ 'title_completion': [ { 'length': 9, 'offset': 0,
  'options': [
```

```
{'score': 0.0020846569, 'text': 'star three'},
{'score': 0.0019600056, 'text': 'star trek'},
{'score': 0.0016883487, 'text': 'star trip'},
{'score': 0.0016621534, 'text': 'star they'},
{'score': 0.0016162122, 'text': 'star tree'}],
'text': u'star trec']}]}
```

系统对每一项建议都做了评价，表示其与用户所期望的查询相匹配的程度。但正如我们在此处看到的那样，这种做法并非总是有效的——star trek 并不是排在最前面的建议项。让我们对此来做一些改进，如清单 8.8 所示。

清单 8.8 在用户的搜索上下文中获取“搜后建议”

```
{ "fields": ["title"],
  "query": {
    "match": {"star trec"}},
  "suggest": {
    "title_suggestion": {
      "text": "star trec",
      "phrase": {
        "field": "suggestion",
        "collate": {
          "query": {
            "inline": {
              "match_phrase": {
                "title": "{{suggestion}}"}]]]]]]]]}}}
```

① 将collation加入主查询中

② 针对每一项建议修改要执行的查询

在清单 8.8 中，我们将建议项包含在了相应搜索的正文内，这样就无须发起两次单独的请求了。而且通过使用 collation 功能①，我们还解决了建议不够准确的问题。利用 collation，搜索引擎执行的是一种被称为 mini-search 的搜索，它依据每一项建议，并将没有匹配的建议项删除。在“collate”那一段，我们指定了搜索应该返回哪些建议项；如果文档与建议项匹配，该建议项就会被包含在建议结果中返回给用户。此处，Elasticsearch 会将特殊的 {{suggestion}} 参数替换成每一项建议的具体文本。

请注意，我们是用 match\_phrase 查询来实现 collation 查询的。这样做是为了对可能的建议项做更为严格的限制，它们必须是存在于索引中的短语。例如，假设我们在②处为 collation 用的是匹配查询（match query），则建议项 star three 就会在返回的建议结果中被保留下来，因为有些文档也会包含 star 或 three，即便它们不包含短语 star three。通过将匹配度较低的建议项从结果中过滤掉，我们确保了向用户提供的建议项只会为其带来有意义的结果。

有关 star trec 查询建议的最终结果如清单 8.9 所示。

清单 8.9 采用 collation 之后的搜索及建议结果

```
{ 'title_completion': [{ 'length': 9, 'offset': 0, 'options': [
  { 'score': 0.0019600056, 'text': 'star trek'},
  { 'score': 0.0016621534, 'text': 'star they'}] ]}]}
```

此处有几点需要注意。首先，请注意由于采用了带 collation 功能的 match\_phrase 搜索，先前的几个建议项已经被我们从结果中去掉了。同时还要注意的，现在排名最靠前的建议项（top-ranking suggestion）已经是与当前搜索最为匹配的建议项了。

另一个麻烦的问题是：应用系统应该如何处置获得的这些建议呢？应该将用户查询完全替换成评价最高的建议项吗？还是将建议项以“你是不是打算（did you mean）”这样的形式呈现给用户呢？遗憾的是，对于这一问题的答案从来都不是那么显而易见的。对由 phrase suggester 所提供的建议项进行评价，我们使用的是一个以文本编辑距离（text edit distance）和被推荐单词的频率（frequency of the suggested terms）为依据的函数。不过仅仅因为一个建议项在某种程度上相比于用户查询“更有可能”是其想要的，就认为用户查询有误，那也未必；用户也许就是在查找某些非常特别的东西。

实际上，我们永远都无法完全读懂某位具体用户的心思。因此，一个好的办法是仔细研究 Elasticsearch 的 phrase suggester 所提供的全部参数，构造一个解决方案，然后观察其执行的效果。不过作为一条经验法则，以“你是不是打算”这样的形式给出建议，引导用户获得更满意的查询，而不是未经允许就将用户搜索替换掉，这样的做法通常而言更为安全。

但是对于这一法则还有一个大的例外：如果用户查询返回的结果为空，则为用户提供可能具有一定相关性的结果，而不是返回给用户完全空的结果，这样的做法总是更有意义一些。同时，无论哪一种情况（搜索替换，或搜后建议），确保 UI 界面能很容易地提示正在发生的情况，从而让用户不至于因为与应用系统的一次交互没有达到预期，而迷失了方向。

## 8.2 浏览期间的相关性反馈

我们已经讨论了在搜索栏内或周围有可能出现的相关性反馈的几个方面。不过，

围绕搜索栏的交互仅仅只是搜索用户和搜索应用之间对话的一小部分。本节我们将把重点放在用户的浏览体验上。

用户可以通过选择性地过滤搜索结果来对其进行浏览，将范围缩小至一个与用户信息需求最为相关的较小集合上来。有的时候，用户甚至可能是直接从浏览文档而非文本查询开始的。浏览体验不同于和搜索栏的交互。与搜索栏的交互为用户提供丰富的相关性反馈，而浏览体验带给用户的则是有关文档在集合中如何分布的一个广阔的视图。这让用户能够就如何对文档进行过滤做出符合其预期的选择。

切面浏览（faceted browsing）是为了方便用户浏览的一种常见方法。浏览几乎任何一家电子商务网站，我们都会见到同样的模式：在页面顶部是搜索栏。在搜索栏下方，并且通常是屏幕的左侧，有一个分类列表。在每个分类下面，是一些可以被选中用来对结果进行过滤的条目。这些菜单项被称为切面（在第2章里我们曾经介绍过）。通常，切面包含一系列匹配某个特定过滤条件的文档。如果用户选择了某个切面，搜索结果就会根据选择条件得到过滤。如图8.5所示，美捷步（Zappos）提供了一个基于切面的浏览界面的极好例子。



图 8.5 “切面浏览”帮助美捷步用户对商品进行分类，并缩小结果集范围，只包含那些与用户搜索目标最为匹配的条目

此处，一条有关“leather sandals”的搜索返回了各种款式的凉鞋——其中女式居多，不过我们也能见到一款童鞋。而用户 John 恰好还处于青少年阶段，因此对这

些搜索结果都不感兴趣。不过他看了一下屏幕左侧的分类,发现可以根据商品类型、性别,以及品牌对这些结果进行过滤。不仅如此,利用性别,商品被分成了4组:女款、男款、少女款和少男款,并且尽管绝大部分匹配 *leather sandals* 的商品都是女款的(有4010种),还是有另外530种是 John 可能会感兴趣的。如果系统还没有为用户呈现这些过滤后的信息,用户可能就会根据商品最初的搜索集合,认为美捷步是没有男款的。如果用户单击“Men”,结果就会得到过滤,只显示男款凉鞋。在此基础上,用户可以通过选择品牌或鞋码,或任何其他条件进一步缩小范围。

试想一下,基于切面的浏览界面为用户提供的大量相关性反馈。通过对侧边栏中信息的快速扫描,用户对各分类商品以及与之相关的元数据就有了深入的理解。通过查看文档数量,用户对这些商品在各分类中的分布情况也会有所了解。最后,也是最重要的一点,用户可以通过单击切面并缩小搜索结果的范围来运用他所掌握的这些知识。在一个电子商务的搜索应用中,客户能够快速浏览数据并找到其所需要的商品,这种能力增加了客户购买商品的可能性。这样的技术也有助于我们规避和简化相关性的处理流程。通过为用户提供更多的选择来引导他们自己找到所需,我们就可以对复杂排名以及试图领会用户意图这样的工作少操点心了。

在下面几个小节中,我们来讨论一下如何在 Elasticsearch 中实现“切面浏览”。我们还将谈到几个与之相关的话题。在 8.2.2 小节,我们会引入一个简单的 UI 功能,帮助用户记住当前其选择的切面。在 8.2.3 小节,我们会讨论用户如何才能更精细地控制相关性排名的优先级。

### 8.2.1 构建基于切面的浏览

我们可以利用 Elasticsearch 的聚合(aggregation)功能来实现切面。为了便于后续讨论,聚合返回的是当前搜索结果中特定分类的计数值。我们假设一个来自 TMDB 的例子——影片分类。所有 TMDB 集中的电影都有一个字段 *genres.name*,包含了像 Adventure、Comedy 和 Drama 这样的标签。下列查询统计了整个 TMDB 数据集中每一类影片的数量:

```
GET tmdb/_search
{
  "aggregations": {
    "genres": {
      "terms": {
        "field": "genres.name"
      }
    }
  }
}
```

它告诉 Elasticsearch 返回一个名为 `genres` 的聚合。针对每一个在 `genres.name` 的全局词典 (`global term dictionary`) 中出现的词, 聚合会返回带有该词的文档数量。在本例中, 我们的词与影片分类相关。聚合的返回结果, 如清单 8.10 所示, 可以被用来在我们的切面浏览界面里生成一个分类菜单。

清单 8.10 针对影片分类的切面统计

```
{ 'aggregations': { 'genres': { 'buckets': [
  { 'doc_count': 7546, 'key': 'Drama' },
  { 'doc_count': 5342, 'key': 'Comedy' },
  { 'doc_count': 3878, 'key': 'Thriller' },
  { 'doc_count': 3753, 'key': 'Action' },
  { 'doc_count': 2623, 'key': 'Romance' },
  { 'doc_count': 2165, 'key': 'Adventure' },
  { 'doc_count': 1981, 'key': 'Horror' },
  { 'doc_count': 1861, 'key': 'Crime' },
  { 'doc_count': 1640, 'key': 'Family' },
  { 'doc_count': 1597, 'key': 'Science Fiction' },
  { 'doc_count': 7479, 'key': 'sum_other_doc_count' }
] } }
```

观察这个列表, 我们可以看到, 集合中数量最多的, 是被标记为 `Drama` 的影片, 然后是 `Comedy`、`Thriller`, 依此类推。除此以外, 查看 `sum_other_doc_count` 值我们会发现, 有 7479 部影片属于 `genres`, 但并未排进前 10。

再次强调, 注意分析所采用的流程这一点十分重要。因为在本章的前面几节里, 我们对文档进行了分词, 这样, 未经处理的原生 `token` (`raw token`) 就可以被返回给用户。对于这个具体的例子而言, 不要使用默认的标准分析这一点很重要, 因为它将输入内容按空格进行分割。这样做会导致 `science` 和 `fiction` 被当作两个不同的分类, 这显然是错误的。

在这条查询中, 我们没有对文档集合做限制, 因此统计数字体现的是所有影片在分类上的分布情况。这样做是没有问题的; 对于还没有指定任何其他条件的用户而言, 这是一个非常不错的数据集。首次使用影片搜索应用的用户, 可以快速审视影片在分类上的分布情况, 从而决定接下来干什么。

但是随着用户继续使用搜索应用, 他们接下来会干些什么呢? 假设用户不打算离开 (总是有可能的), 他们有两种选择: 通过搜索框进行搜索, 或者单击切面缩小搜索范围。无论哪种情况, 其有效性结果都是一样的; 他们将具备相关性的文档缩小到了文档集的一个子集。使用聚合的好处在于, 针对过滤后的结果集, 每个切面都会得到统计。



那么，我们假设用户单击了 Science Fiction 这个分类，表明他只想查看标记为 Science Fiction 的影片。然后，我们就可以对结果集进行相应的过滤了：

```
GET /tmdb/_search
{"query": {
  "bool": {
    "filter": [{
      "term": {
        "genres.name": "Science Fiction"
      }
    ]
  }
},
"aggs": {
  "genres": {
    "terms": {"field": "genres.name"}
  }
}
```

有了这个额外的过滤条件，新的搜索结果里就会包含与标记为 Science Fiction 的影片相关的，更新了切面统计结果：

```
{ 'aggregations': { 'genres': { 'buckets': [
  { 'doc_count': 1597, 'key': 'Science Fiction' },
  { 'doc_count': 753, 'key': 'Action' },
  { 'doc_count': 502, 'key': 'Thriller' },
  { 'doc_count': 466, 'key': 'Adventure' },
  { 'doc_count': 337, 'key': 'Drama' },
  { 'doc_count': 336, 'key': 'Fantasy' },
  { 'doc_count': 327, 'key': 'Horror' },
  { 'doc_count': 299, 'key': 'Comedy' },
  { 'doc_count': 188, 'key': 'Animation' },
  { 'doc_count': 164, 'key': 'Family' },
  { 'doc_count_error_upper_bound': 0,
    'sum_other_doc_count': 361 }
] } }
```

不仅切面的统计数据有了变化，搜索结果也得到了更新，其内容只包含科幻影片。走到这一步，用户还会做些什么呢？他们想做什么都行。用户有可能会选择其他小的分类，在同一个大类中再次进行过滤。也有可能基于其他条件进行过滤，比如发行日期。用户一旦知道了自己想找的是什么，他们甚至可能会抛开浏览交互，转而使用文本框。不管是哪种情况，对切面的统计数据以及在搜索结果中出现的文档，都会有助于引导客户搞清楚能查到什么，以及去哪里查。

### 8.2.2 提供面包线导航

我们时常希望就用户对数据的过滤情况为其提供反馈信息。如果没有反馈，用户也许就会因为没有意识到还有某些过滤条件在起作用，而惊讶于为什么没有搜到任何内容。面包线导航是一项广泛用于防范此类问题发生的技术。如图 8.6 所示，

典型情况下，面包线导航会以当前被选中的一组切面的形式，被置于搜索结果的上方。此处，我们依然选择前面美捷步的那个搜索“leather sandals”的例子。这一次我们将结果集进一步过滤到男款、价格在 \$100 以下的棕色鞋子。

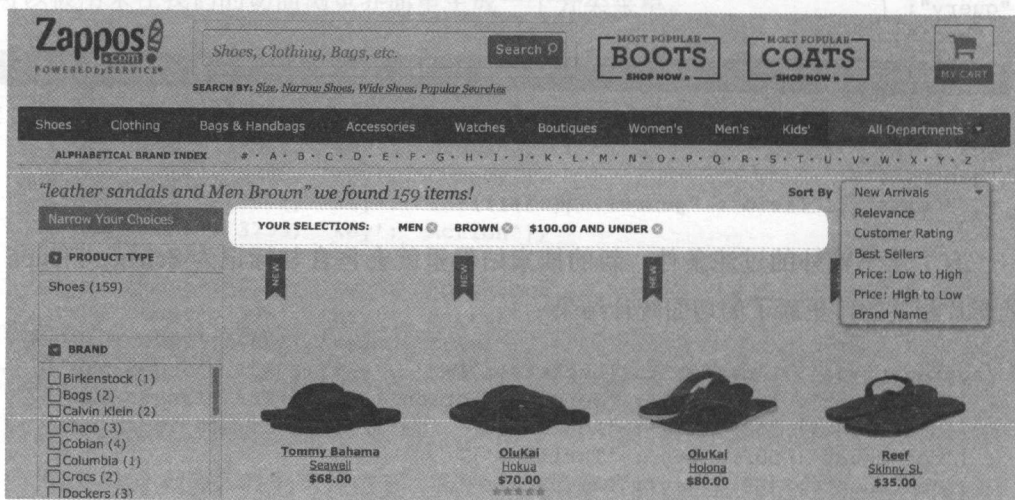


图 8.6 美捷步利用面包线导航来提醒用户当前选择的切面，并允许他们很容易地取消这些选中的过滤条件。

正如我们看到的那样，面包线导航不仅提示用户选择了哪些过滤条件，而且作为一种直观的界面，还为用户提供了从中去掉部分过滤条件的能力。

### 8.2.3 选择其他的结果排序方式

在以浏览方式进行搜索的过程中，什么才是最适合的结果集排序呢？本书通篇都在讨论如何根据相关性来呈现结果。但在用户尚未指定查询文本的时候，相关性的意义并不是很大。相反的，我们得由搜索应用自己来决定默认的排序规则。取决于具体领域，默认排序可以基于热门程度、位置远近、时效性，以及许多其他因素。对默认排序的选择甚至有可能是搜索应用植入商业诉求的一个很好的机会，比如，在不知不觉中将高利润商品推高置顶。

我们也可以赋予用户控制结果排名的权力。就如同简单地在结果集的最上方放一个下拉选择框一样。这样做通常会让搜索结果越来越准确，再次以美捷步搜索为例，如图 8.7 所示。

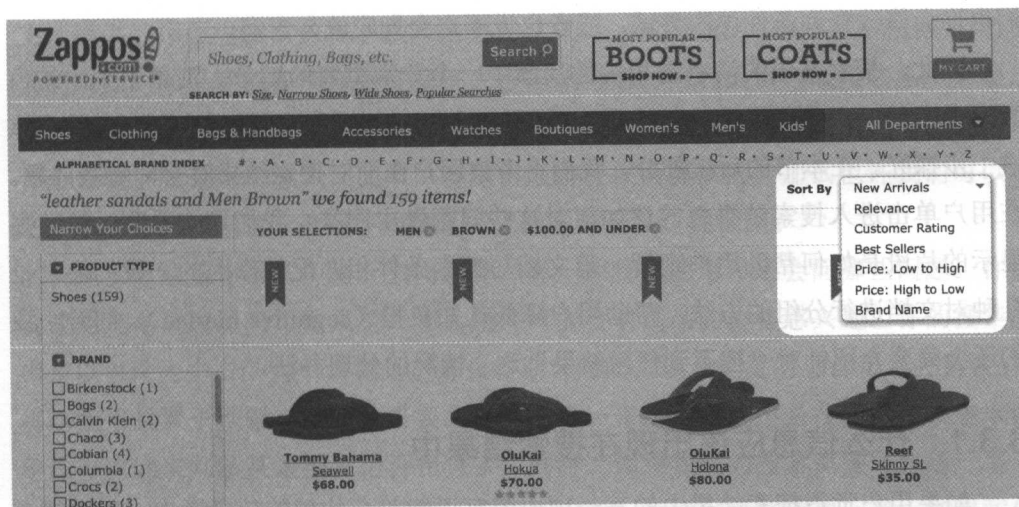


图 8.7 美捷步允许用户根据不同条件对结果进行排序

美捷步允许客户根据新品上架、用户评分、热门程度，以及价格高低，对结果进行排序。允许用户控制排序条件是相关性反馈的一种绝佳的表现形式。它让用户对结果有更准确的理解，并为用户提供了有效组织搜索结果，匹配当前搜索目标的手段。

也有可能我们并不总是希望根据前述这些因素直接对结果进行排序。用户要求根据“热门程度”排序，可能只是希望热门商品的排名能够靠前一些：但这一因素的影响还不至于大到盖过用户的其他搜索条件。例如，假设有一位正在搜索潮款“leather sandals”的用户，搜到了一堆热销的人字拖（flip flops），那他也许是不会感到满意的。看一下第 7 章有关放大处理（boost）的部分，我们是否应该以放大处理的方式来实现排序功能呢？

## 8.3 搜索结果清单中的相关性反馈

在本章中，我们根据用户使用搜索应用的典型途径，从搜索框中的交互开始谈起，一直谈到了基于切面搜索的浏览与过滤。这些技术为用户获取相关性内容提供了许多选择。最后，让我们来谈一谈针对搜索结果本身的相关性反馈与指导原则。在这里，用户得到的是一个被搜索应用认为满足相关性要求的文档摘要的清单。如果用户对其中某项搜索结果感兴趣，他们会单击进去并继续查看，以期能从中找到他们想要的内容。如果用户瞥了一眼搜索结果之后发现并没有找到想要的内容，他

们也许会修改查询条件再次尝试，或者也说不定会彻底放弃查询。

因此，搜索结果的清单是相关性反馈中一个非常重要的部分。清单中列出的内容所反映的，必须是匹配文档中最为重要的部分，以使用户能够做出判断，是否该进一步探究。在本节的后续部分，我们将考察用户在面对搜索结果时会问哪些问题。在用户单击进入搜索结果之前该如何对这些问题做出回答。我们还将看到，被高亮显示的片段是如何帮助用户理解一篇文档与搜索条件相匹配的原因的。我们会讨论几种对文档进行分组的方法，帮助用户降低认知负担（cognitive burden）。同时，我们还会简要介绍针对“搜不到任何结果”这一情况的处理办法。

### 8.3.1 什么信息应该出现在搜索结果中

如果用户觉得搜索结果中的某一项内容有可能符合他们的信息诉求，用户就会单击进去。因此我们必须考虑到，用户在审视搜索结果时会问及什么样的问题。呈现在搜索结果中的信息应当尽可能完整而简明地对这些问题做出解答。

在众多要解答的问题当中，首当其冲的是——“这是什么”；如果运气好的话，回答这一问题也许只是简单地将文档标题放入搜索结果。但是标题时常是文档的一种不可靠的语义表述。例如，在企业领域里，搜索引擎所掌握的也许是那些官方的表格和文档，其标题晦涩难懂，比如 Form I-130 或 401(k)，并没有轻易展示出表格的意图。同样的，在电商领域，商品被赋予的标题时常是为了夺人眼球和传递感观，却未必揭示了商品的准确含义。例如，Oakley 有一款有名的女式太阳镜系列，名叫 Little Black Dress——就是一个很搞笑的名字。

如果我们发现自己身处这样一个文档标题意义不明的境地时，就必须找到其他办法来为用户提供信息。为了回答“这是什么”，在电商领域里，图片时常要比标题（或者其他任何文本，就这一场景而言）来得更直接。一位寻找某件商品的用户可以快速浏览页面上的图片，并迅速缩小需要着重关注的范围。甚至企业的文档检索也可以通过将图片纳入搜索结果这样的手段从中获益。例如，在搜索结果里包含一张小幅的文档扫描件，也许会有助于用户准确地找到他们过去曾经见过和用过的文件。

简短的说明也有助于为用户解答“这是什么？”的问题，但此处需要格外注意。由于有太多的文本在吸引用户注意力方面是彼此相互竞争的，这些文本有可能会丢失重要的细节，或者还要更糟：被大量的信息所淹没，完全妨碍了搜索。不过假如

文档提供的标题无法有效揭示搜索内容的含义，一段简短的说明也许就是我们所需要的东西。如果标题和简介意义都不大，那就考虑与内容提供方取得联系，并要求提供新的简介。如果被搜索的内容是由用户提供的，也许我们可以为文档简介提供一个必填字段让用户输入。如果内容是由上游提供商提供的，那么也许我们可以与之合作，共同制作这一点额外的信息，让他们的产品更加畅销。

除了“这是什么？”这一大问题外，想一想我们的用户还会问哪些问题。你所构建的是电商应用吗？如果是，那么用户可能会对商品的价格感兴趣。你的搜索应用是以事件为中心的？如果是，那么日期、时间，以及位置信息都是至关重要的。将这些信息置于搜索结果中的显要之处，这样一来，用户就不必在搜索结果中来回单击，以此来判断某条搜索是否相关了。同时，如果我们的搜索范围还包括一系列子集，那就也考虑一下用户在这些子集里搜索时，会遇到哪些重要问题。如果我们知道用户正在购买相机，就把涉及相机的有用的详情信息放到搜索结果里，这样用户自己就能在搜索页面里对各种选择进行比较了。

### 8.3.2 通过文本片段与高亮提供相关性反馈

当搜索的领域涉及包含大量文本的文档时，文本片段（snippets）与高亮（highlighting）为我们提供了相关性反馈的一种重要形式。也许你已经猜到了片段与高亮的意思。如果用户提交了一条关键词搜索，所谓片段就是从匹配文档中摘取出来的、包含搜索关键词的一段段文字。而高亮则是出现于文本片段中的关键词匹配，以突出显示的方式呈现给用户。为此我们可以使用粗体显示文本或者调整关键词匹配处的背景色。有的搜索界面走得更远，它们为每个关键词选择不同的高亮风格。

片段高亮的相关性反馈，其目的是要告诉用户，为什么一篇文档与用户查询相匹配，以及匹配在什么地方。用户，尤其是那些深谙搜索的高手们，会非常希望自己能在此上下文中快速检视匹配出现的地方。在不必单击进入某项搜索结果的情况下，用户就能了解文档是否与其信息需求相符。与此同时，用户还可以在此基础上查看存在于匹配文本中的模式，并选择进一步细化他们的搜索，以获得合适的匹配文档。有时，针对用户问题的答案也许在文本片段中即刻就能找到，从而，用户甚至都不必去单击和查看发生匹配的文档。

在本书中，我们已经讨论过有关文本分析要考虑的方方面面。在第4章中，我们展示了如何构造分析方法，捕获文档中单词的语义理解。而在本章的许多不同地



方，为了支持在切面和建议项里使用易于人类理解的 token (human-readable token)，我们似乎又将这一观点推翻了。借助高亮，我们可以放心地回到利用 token 捕获语义的这一方法上来。换言之，经过高亮的 token 不必一定要易于人类理解。让这一结果成为可能是基于这样一个事实，Elasticsearch——或者更准确地说，是 Lucene——在进行分词之前记录了原来单词开始与结束字符的偏移量。在高亮处理时，Lucene 可以根据 token 进行匹配，查找原有字符的位置，并将高亮置于事先保存的文本中合适的地方。

Elasticsearch 提供了三种高亮模式，各自有不同的权衡取舍：

- 基本高亮 (basic highlighter, 默认模式)
- posting 式高亮 (postings highlighter)
- 快速向量式高亮 (fast vector highlighter)

basic highlighter (默认模式) 不要求有专门的设置；我们只是简单地要求 Elasticsearch 在返回搜索结果的同时带上高亮片段。随后经过高亮处理的文本片段就会与搜索结果一起被返回。不幸的是，为了找到文档中与搜索条件相匹配的词的位置，这一默认实现必须重新对文档进行分析。对于小篇幅文档而言，重新分析所需的时间是微不足道的。但对于篇幅较长的文档而言（比如数页篇幅的文字），这种重复处理带来了相当显著的性能开销。

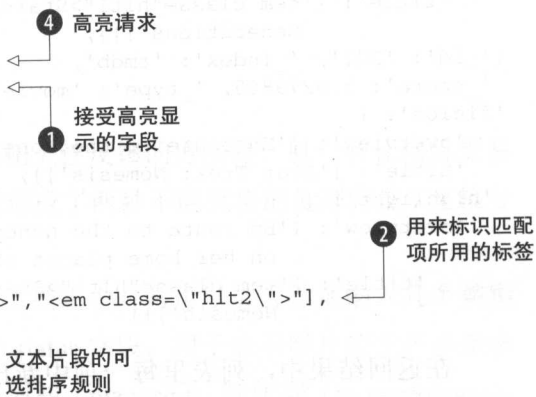
postings highlighter 与 fast vector highlighter 通过在索引期间存储额外信息，避免了查询期间的处理 (query-time processing)。这种做法带来的代价是索引规模的增加。不过这两种方法也有一定的优势。例如，postings highlighter 对自然语言文本 (natural language text) 尤其有用（句子和段落，与标题或由单个词所构成的字段相对）。它自动将输入文本进行切分，以便将文本片段以完整句子的形式返回给用户。fast vector highlighter 可以对匹配的词进行独立的高亮处理，允许每个被高亮的关键词有其自己的颜色或风格。

为了演示片段高亮的运用，清单 8.11 展示了其中一类 highlighter，即 fast vector highlighter。该清单向我们展示了一些有意思的功能，使高亮成为相关性反馈的一个非常重要的来源。



## 清单 8.11 提交查询并开启高亮功能

```
GET tmdb/_search
{
  "fields": ["title", "overview"],
  "query": {
    "match": {
      "title": "star trek"
    },
    "highlight": {
      "fields": {
        "title": {
          "number_of_fragments": 0,
          "overview": {
            "fragment_size": 100,
            "number_of_fragments": 5,
            "no_match_size": 200
          }
        }
      },
      "pre_tags": ["<em class=\"hlt1\">"],
      "post_tags": ["</em>"],
      "order": "score"
    }
  }
}
```



在提交查询之前，我们对文档进行索引时，同时为 title 和 overview 字段开启了词向量（term vector）。（为此，我们在字段映射里加上了 "term\_vector": "with\_positions\_offsets"。）现在让我们看一下该查询中的某些细节部分。

为了开启高亮功能，我们在其中新引入了一段与高亮相关的内容<sup>④</sup>。在“highlight”定义的 fields 部分<sup>①</sup>，我们指定了希望高亮的字段，以及任何与该字段相关的我们所需的其他参数。（稍后我们会更详细地讨论这些参数设置。）可选参数 pre\_tags 和 post\_tags<sup>②</sup>指定了如何在结果中对经过高亮的关键词进行标注（annotate）。通过指定若干标签（tag），我们可以为每个关键词设置不同的风格。例如，我们可以为每个关键词设定自己的颜色。这样可以让用户快速查看搜索结果并立刻理解为什么给定文档与他们的查询相匹配。最后一个参数 order<sup>③</sup>对于涉及大篇幅文档的搜索特别有用。根据评价值对片段进行排序，使高亮处理顺带对片段做了排序，从而优先返回最为匹配的片段。

既然我们已经开启了高亮功能，那就让我们来看一段针对“Star Trek”的查询结果吧，如清单 8.12 所示。

## 清单 8.12 开启高亮功能后，一次针对“Star Trek”的搜索返回的部分内容

```
{ '_id': '193', '_index': 'tmdb',
  '_score': 5.0279865, '_type': 'movie',
  'fields': {
    'overview': ['Captain Jean-Luc Picard ...'],
  }
}
```

```

    'title': ['Star Trek: Generations']],
    'highlight': {
      'overview': ['renegade scientist Soran who is destroying entire
        <em class="hlt1">star</em> systems. Only one man
        can help Picard stop Soran\'s'],
      'title': ['<em class="hlt1">Star</em> <em class="hlt2">Trek</em>:
        Generations']]],
    {'_id': '201', '_index': 'tmdb',
      '_score': 5.0279865, '_type': 'movie',
      'fields': {
        'overview': ['En route to the honeymoon of William Riker ...'],
        'title': ['Star Trek: Nemesis']},
      'highlight': {
        'overview': ['En route to the honeymoon of William Riker to Deanna Troi
          on her home planet of Betazed, Captain Jean-Luc'],
        'title': ['<em class="hlt1">Star</em> <em class="hlt2">Trek</em>:
          Nemesis']}]

```

在返回结果中，列表里每一项中新出现的 `highlight` 部分都被标成了粗体。首先我们注意到的是，要感谢 `fast vector highlighter` 的使用，以及指定的 `pre` 和 `post` 标签，关键词 `star` 和 `trek` 被封装在了不同的标签里。

现在，让我们回到清单 8.11，从那些字段级的高亮设置中选择一两个进一步做分析。对于短文本字段，比如 `title`，我们不希望只返回一小段文本；而是整段文本。通过指定返回的 `number_of_fragments` 为 0，我们告诉了 `Elasticsearch` 对字段的全部文本进行高亮处理并返回。接下来，对于 `overview` 字段，我们可以看到，片段的长短和数量可以相应地通过参数 `fragment_size` 和 `num_of_fragments` 来进行指定。

下一个参数，`no_match_size`，可能会非常有用。一篇与搜索完全匹配的文档和我们用于进行高亮处理的字段不存在任何匹配，这种情况是经常发生的。典型情况下，这就意味着不会有针对该字段的片段被返回。不过，与其什么都不返回，我们为什么不至少返回一点字段里开头部分的文本呢。这样，用户就能获得一定的上下文信息了。这就是参数 `no_match_size` 的用武之地。此处，当我们指定 `no_match_size` 为 200 后，任何在 `overview` 字段里没有关键字匹配的文档，都将返回前面大约 200 个字符，最大限度地匹配整个单词。请注意，这恰好是清单 8.12 中的第二个文档的情况。由于 `star` 和 `trek` 都没有出现在该字段里，所以我们就返回了开头一段文本。

关于清单 8.12 还有最后一点要注意：文本片段的切分位置是随意的。如果我们选择 `postings highlighter` 而不是 `fast vector highlighter`，那么文本片段的切分就会以

句子为单位，通过生成更易于理解的文本片段，这样做无疑提供了更好的用户体验。不过 `postings highlighter` 无法区别不同的关键词，因此我们无法将之以不同形式呈现在搜索结果中。由于这样的情况时有发生，所以，我们在选择实现方案时必须透彻理解，并仔细权衡利弊。

### 8.3.3 对相似文档分组

另一种在搜索结果列表中为用户提供相关性反馈的方法是，以一种让信息更易于用户处理的方式对文档进行分组。这就形成了两种不同风格的处理方法：文档分组与字段合并。

我们首先来看一下文档分组，文档在文档集中时常可以被天然地纳入几个类别中。例如，TMDB 数据集中的影片有一个 `status` 字段，用于表示影片位于产品流水线上的哪个阶段：造势期（`rumored`），筹划期（`planned`），制作期（`in production`），以及发行期（`released`）。对于某些可能的搜索应用而言，将搜索结果根据已有分类预先进行分组然后加以呈现，也许是很有帮助的。如果我们的用户首先考虑的也是文档的分组情况，那么这一功能就可以极大地降低他们的认知负担了。

对文档的分组可以使用聚合来实现——在本例中，即 `term` 聚合与 `top hits` 聚合的组合，参见清单 8.13。

清单 8.13 使用 `term` 及 `top hits` 聚合的组合为文档进行分组

```
GET /tmdb/_search
{ "query": {
  "match": {
    "title": "star trek"
  },
  "aggs": {
    "statuses": {
      "terms": { "field": "status" },
      "aggs": {
        "hits": {
          "top_hits": {}
        }
      }
    }
  }
}
```

这条查询是让 Elasticsearch 找出所有匹配 `Star Trek` 的文档，然后根据状态进行分组。相应的结果集会包含一段 `buckets` 信息，如清单 8.14 所示。

清单 8.14 根据 `term` 和 `top hits` 聚合的组合对文档进行分组

```
{ 'buckets': [
  { 'doc_count': 82,
    'key': 'released',
```

```

    'hits': { 'hits': { 'hits': [
      { '_id': '13475',
        '_index': 'tmdb',
        '_score': 6.032624,
        '_source': {
          'name': 'Star Trek: Alternate Reality Collection',
          'popularity': 2.73003887701698, ... },
        /* more documents */ },
    { 'doc_count': 4,
      'key': 'in production',
      'hits': { 'hits': { 'hits': [
        { '_id': '13475',
          '_index': 'tmdb',
          '_score': 6.032624,
          '_source': {
            'name': 'Star Trek: Axanar',
            'popularity': 3.794237016983887, ... },
          /* more documents */ },
        /* more groups */ },
      'doc_count_error_upper_bound': 0,
      'sum_other_doc_count': 0]

```

关于分组排序有一点要注意：默认情况下，term 聚合是根据匹配文档的数量以倒序方式排列的。如果默认的顶层排序不适合我们的应用，可以花点时间通读一下 Elasticsearch 的文档，看看我们自己是否能够为顶层的分组构造一个更为适合的排序。（清单 8.15 给出了一个针对顶层分组的自定义排序的例子。）

另一种常见的文档分组形式被称为字段合并（field collapsing）。假设我们有彼此近似的若干文档。例如，假设 TMDB 集合中包含“星际迷航”电影的多个不同的条目，影片被翻译成多种语言，每个条目对应一个语种，我们该如何处理这种情况呢？与其让用户的搜索结果里，每一页都充斥着实际内容相同的结果，我们可以利用字段合并来对接近重复（near-duplicate）的文档进行分组，然后只为用户返回该分组集合中的一个样例文档（exemplar document）。

为了构造这样的搜索返回结果，我们再次结合使用了 term 和 top hits 聚合。弄清楚细节需要花一点时间。为了实现字段合并必须要做一点配置：我们需要一个字段来指定哪些文档是彼此相近的。TMDB 数据集并没有一个现成的例子，让我们再次以“星际迷航”为例，假设影片有多个不同的条目，每个条目对应影片被翻译为一个语种。更进一步，假设不论哪种语言，每个条目都包含一个全局唯一的标识，指向英文原版。这个全局唯一的标识就是进行字段合并的一个很好的候选字段。通过将所有在该字段上取值相同的影片组合在一起，我们就可以成功地将具有多个

翻译版本的影片合并成一个了。

接下来的一个难点是修改顶级分组的排序。回忆一下，顶级分组默认情况下是根据分组所包含的文档数量进行排序的。在本例中，顶级分组对应原版影片，并且我们依然希望搜索结果能按照相关性进行排序。为了达到这一效果，我们根据每个分组中相关性最好的文档的评价值，对顶级分组进行排序。清单 8.15 与清单 8.13 很类似，只不过我们对正确实现字段合并所做的必要改动做了突出显示。

清单 8.15 根据分组中相关性最好的文档进行的分组排序

```
GET /tmdb/_search
```

```
{
  "query": {
    "match": {
      "title": "star trek"
    }
  },
  "aggs": {
    "original_versions": {
      "terms": {
        "field": "original_id",
        "order": { "top_score": "desc" }
      },
      "aggs": {
        "hits": {
          "top_hits": { "size": 1 }
        },
        "top_score": {
          "max": { "script": "_score" }
        }
      }
    }
  }
}
```

① 根据original\_id  
字段进行合并，  
并按top\_score  
排序

具体而言，顶层的 term 聚合现在指向的是 original\_id 字段；term 聚合是根据 top\_score 进行排序的；紧随 top hits 聚合后面的 top\_score，其定义为：与每个顶层 bucket 相关联的文档中的评价值最大者。

### 8.3.4 在用户搜不到结果时给予帮助

有时，用户发起的请求会得不到任何结果。出现这种情况，有可能是因为用户的疏忽，比如在搜索请求中出现了拼写错误，也有可能是因为用户对搜索请求限制太多，以至于没有文档与之相符。在这种情况下，我们能做的最糟糕的事情是，给用户返回一个空的搜索结果页面。这样做除了令用户放弃搜索转而寻找其他途径来满足其需求外，也使其陷入了孤立无助的境地。

为了预防此类事情的发生，我们始终要有备选方案，以便尽可能地满足用户的搜索目标。在 2.1.3 节，我们讨论了如何使用建议项来纠正用户的错误。如果没有搜到任何结果，我们可以在此基础上更进一步，将用户的查询替换成最佳建议项。或

者，如果用户一方没有明显的错误，也许我们可以自动删除用户的最后一个过滤条件，这样，搜索条件就可以更宽松一些了。最后，如果我们实在没有别的办法，那就将最受欢迎的文档返回给用户。无论是哪种情况，如果搜索应用替用户对搜索做了修改，请明确告知用户搜索已经被改变了，这样用户就不会因为搜索结果与预期不符而被搞得晕头转向。

## 8.4 本章小结

- 相关性反馈为搜索应用与用户之间的交互提供了便利。
- 在涉及搜索体验时，要考虑用户的典型流程。他们通常从文本搜索开始，然后在查看搜索结果时，会引入条件过滤并对查询条件进行细化。最后，找到感兴趣的内容并单击进入详情页面。
- 通过在用户输入时更新搜索结果，帮助用户更快地找到所需内容。
- 通过为“搜索补全”提供建议，帮助用户构造成功的搜索查询。
- 利用“你是不是打算 (did you mean)”这样的建议来纠正用户的错误。
- 基于切面的搜索帮助用户理解索引中内容的分布情况，允许用户过滤并向下钻取 (drill down)，直至其所期望的搜索结果的子集。
- 面包线 UI 组件可以让用户明白其搜索结果是如何被过滤的。同时它也为用户提供了一种直观的去掉现有过滤条件的方法。
- 经过高亮处理的搜索结果和详情页面可以让用户注意到，为什么搜索引擎认为某一条结果是重要的。
- 对结果的分组和排序，通过让用户关注最引人注目的搜索结果，减少用户的认知负担。
- 相关性反馈让用户更易于介入搜索活动，并找到他们所需要的内容。



# 设计以相关性为核心的搜索应用

## 本章要点

- 在构建一个新的搜索应用之前进行信息收集
- 设计与实现一个完整的搜索应用
- 设计能将子查询组合起来的复合查询
- 在查询参数之间保持均衡
- 搜索的部署、监控和改进
- 知道什么时候进一步的相关性调优不会再起作用

在前面几章中，我们阐述了良好的搜索应该具备的全部要素：

- 通过正确的分词从文档的文本中提取特征。
- 定义重要的信号，并构造搜索字段来代表这些信号。
- 建立同时兼顾用户需求和业务需求的查询。
- 向用户提供反馈以指导他们得到更为相关的结果。

现在，本章将完成余下的部分——告诉大家如何将这要素组织起来，并从方法论的角度阐述如何构建一个以相关性为核心的搜索应用。

在此之前，我们已经深入探讨了搜索引擎本身的一些细节，并介绍了如何把相关性搜索的体验带给用户。在这一章（以及后面的章节）中，我们将把视角从底层的细节转向上层的搜索应用开发。本章，我们将介绍构建搜索应用的一种系统化方法，其基础是图 9.1 所示的这样一个简单的流程。从上层的视角来看，设计一个搜索应用需要三个步骤：搜索需求的收集；应用的设计；对已部署应用的维护。在接下来的部分，我们对每一步都做了详细的解释，以便大家在读完本章之后，对于如何构建自己的搜索应用能有一套坚实的系统化思维。

和以往所有章节一样，我们会用一个有趣的例子来推动讨论的进行。从“星际迷航（Star Trek）”的例子中，我们已经挖掘出了所有值得讲解、而又生动有趣的内容，这次换一个别的例子。我们将开始打造一款激动人心的全新应用，名叫 Yowl，搜索将是它的核心功能。

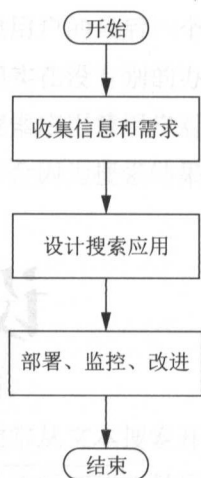


图 9.1 相关性搜索由三个步骤构成：信息收集、设计，以及部署。本章会详细介绍这三个步骤。

## 9.1 Yowl！一个绝佳的新起点

先说最重要的，我们有一个棘手的问题需要着手解决，如图 9.2 所示。

幸亏 Doug 有一个好点子！试想一下：你下了飞机来到一座以前从未到过的城市，在过去的 5 个小时里你只吃了一点饼干和姜汁饮料，现在非常渴望吃一点能填饱肚子的东西——一个又大又多汁的汉堡。但我们如何才能找到一家不错的汉堡连锁店呢？市面上还没有哪一款产品能够帮我们在附近找到满意的餐厅。

Yowl 可以急人所难！通过 Yowl 的智能手机应用，我们就能搜索并快速找到附近满足要求的餐厅。又或者，你也许并不是在一个陌生的城市里，而是热衷于寻找自己所在城市的美食。Yowl 可以帮助我们找到自己从未去过的餐厅，找到能完全满足我们口味的餐厅，甚至还会帮助我们发掘出新的口味！一切皆有可能。



图 9.2 任何一个良好的搜索应用，其出发点都是一个待解决的问题。找到问题并认真思考是构建搜索应用的重要一步。

尽管可能需要事先花一些力气，但我们希望随着 Yowl 的日渐流行，所有的数据都将来自餐厅的顾客和老板。餐厅的顾客会对他们就餐的餐厅发表评论，而餐厅的老板也会因为 Yowl 能够带来新的客流而主动更新餐厅的数据，从而使他们的餐厅可以被轻易找到。

最棒的就是，Yowl 在这个市场上完全没有任何竞争对手。很惊讶是吧？怎么之前就没有人想过这个呢？Doug 说他在 Google 上搜过这个点子，没有查到！嗨，各位，一旦我们打造出这款新的搜索应用，我们就能成富翁了。哦，最后解释一下这个想法的取名；你可能会觉得 Yowl 就像是当你踩到猫尾巴时猫发出的声音。或许是吧。但我们认为这个名字朗朗上口。把 Yowl 想象成当你饥饿难耐的时候，肚子咕咕叫的声音吧。这样的话，当你饿得前胸贴后背的时候，就会立即想起我们这个 App 啦！

## 9.2 信息和需求的收集

在构建搜索应用之前，让我们花点时间考虑一下所有利益相关者的需求和期望。这是我们这一流程中紧接着要做的事情，如图 9.3 所示。这其中最显而易见的一点是，搜索应该易于使用并对用户有所帮助。不过 Yowl 是一个搜索驱动的商业应用，因此它也必须满足商业需求。在本节中，我们同时来看一看用户和商业两方面的需求。找出满足这些需求所需的信息，并对这些信息潜在的来源加以描述。

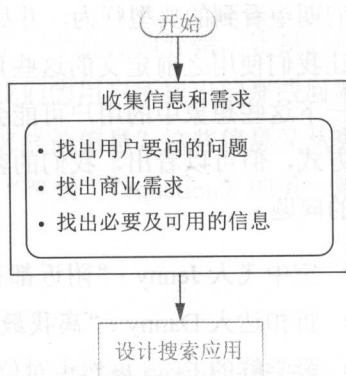


图 9.3 在构建搜索之前，考虑所有参与者的需求，找出实现需求所需的信息，并确定从哪里能找到这些信息。

### 9.2.1 理解用户及其信息需求

为那些预计会和应用有所交互的各种不同类型的用户创建相应的角色代表 (persona)，这是一项很有用的练习。角色代表描述的是一个概括了某一类用户行为的虚构人物。以下是我们对 Yowl 用户角色代表的初步印象。

- 空中飞人 Jenny——Jenny 经常旅行，每到一个新的城市，常会使用 Yowl 来查找就餐的地方。有时她会寻找一个快捷方便的地方随便吃一顿；有时她会

找一个高档餐厅和商业伙伴一起共享美食。

- 美食家 Courtney——Courtney 追求生活中美好的事物，对于食物而言，她有着苛刻的品味。今天晚上她会去造访最好的法式菜肴，说不定哪天晚上，她又会去寻找有最好的散寿司（Chirashizushi，一种日本菜）可供售卖的餐厅。
- 爱尝鲜的 Evan——Evan 兴趣广泛，他总是乐于尝试新鲜事物，包括新的美食品种、新的餐厅，和城市里新开发的地方。
- 折扣达人 Danny——Danny 对价格比较敏感。在选择餐厅的时候，他希望找到性价比最好的那一家。他会寻找价格低但口碑好的餐厅。如果有折扣或特价菜，他就会选择。他更喜欢就近的餐厅——为了省汽油，你懂的。

对用户角色代表的构造可能乍看起来会有点武断，但这些角色代表可以让我们从更具体的层面把用户区分开来。角色代表也为我们形成了一套共用的词汇，从而方便我们讨论用户行为的整体分布。当为项目创建角色代表时，请确保其涵盖了所有我们期望看到的典型行为，并尽量避免行为的交叠。

让我们使用之前定义的这些角色代表来了解一下我们的用户。为此，我们可以列举一下这些想象中的用户可能会对我们的服务所提出的问题。虽然有各自不同的表达方式，但可以看出，我们的客户中有三位对餐厅的位置感兴趣。他们可能会问下面的问题：

- 空中飞人 Jenny：“附近都有哪些餐厅？”
- 折扣达人 Danny：“离我最近的折扣餐厅在哪里？”
- 爱尝鲜的 Evan 虽然也对位置感兴趣，但他感兴趣的不是“附近”或“最近”的餐厅，而是一个稍有不同的问题：“什么地方有新开的热门餐厅？”

接下来最受大家普遍关注的问题似乎就是用户对某种菜肴的口味了。问题如下：

- 空中飞人 Jenny 对菜品可能只要求一个笼统的表达。她可能会问，“我正要和同事出去吃晚饭，附近有哪些意大利餐厅？”或者在一个陌生的城市里，空中飞人 Jenny 可能想找一家熟悉的连锁餐厅，因为她知道在那里可以吃到些什么：“我的肚子已经迫不及待想来点便宜的墨西哥餐了。附近有没有墨西哥肉卷？”
- 美食家 Courtney 需要 Yowl 对菜品的理解要像一位专业厨师那样敏锐。她不会搜索中餐；而是会问更加具体的问题：“本市最好的四川餐馆在哪里？”

这说明我们或许需要对菜品按层级进行管理——例如，湘菜和川菜都是中餐的一个子类。但 Courtney 作为一名真正的美食家，她甚至有可能会问到某些更为具体的菜名：“我在哪里可以找到一种叫棒棒鸡的川菜？”

- 爱尝鲜的 Evan 涉猎广泛，他不会用某一种菜品来约束搜索的结果，但他会对有哪些菜品在售感兴趣。他会问一些探索性的问题，比如“在城市的这一片区域里都有哪些餐厅？”

最后，所有的角色代表似乎都会用额外的条件来对查询结果进行筛选：

- 爱尝鲜的 Evan 感兴趣的是“有什么新菜品？”
- 美食家 Courtney 感兴趣的是“哪样菜的评价高一些？”
- 折扣达人 Danny 感兴趣的是“哪些菜既便宜又好吃（性价比高）？”

Danny 可能还特别想知道都有哪些折扣。嗯……或许我们甚至可以从商业角度出发对折扣加以利用。

找出典型用户及其信息需求，这为我们提供了一个良好的开始，让我们可以识别出能够满足这些需求的必要信息。再回顾一下我们的用户所提出的那些问题，并思考一下：为了回答用户的所有问题，我们需要怎样的信息？这些信息又从哪里得来呢？在弄清楚这些问题之前，让我们来看一看“等式”（equation）的另一端：业务本身的需求。

### 9.2.2 理解业务需求

如果 Yowl 打算长期服务用户，就需要一个可持续的商业模型；需要有人为我们所提供的服务付费。小菜一碟——我们这里有一个很好的点子：为了成为 Yowl 的推广餐厅，这些餐厅将支付一笔会员费。任何时候，只要用户搜索的一类菜系或一道菜品与一家推荐的餐厅相匹配，这家餐厅就会在其相关性评价中获得额外的放大（boost），以确保其在搜索结果中的排名更加靠前。在此基础之上，我们还会对那些提供折扣的推广餐厅做进一步的放大。

对于业务而言，还有一些因素是与钱无关的。Yowl 是一家内容驱动的公司，总的来说，我们依赖于餐厅来提供信息。因此，任何时候只要餐厅提供给 Yowl 的信息是有价值的，Yowl 就会临时对这家餐厅的评价采取一定的放大，以感谢其为我们提供的帮助。

请注意，此处我们还需要对几个彼此竞争的因素做出权衡。如果我们为了迎合餐厅而进行过度放大，就会因为查询结果的相关性失准而失去用户。而如果我们为了讨好用户而进行过度放大，没有给推广的餐厅带来收益，很快就会失去这一部分客户。相关性工程真是复杂啊！

### 9.2.3 找出必要及可用的信息

前面我们已经同时考虑到了用户需求和业务需求，接下来让我们把这些内容再概括一下。我们现在需要回答的三个主要问题是：

- 什么信息是必需的？
- 这些信息会被用来做什么？
- 如何得到它们？

回答了这些问题之后，我们对应用就会有一个更为清晰的理解。就可以找出那些由于信息难以获得或者无法回答用户所提问题而带来的可行性风险。

让我们先来考虑一下搜索的对象——餐厅。最起码，一家餐厅要有一个名字和一个地址。一般这些信息都是可以公开获得的。我们可以从商业数据库提供商那里购买到更加精准，且可以被机器读取的餐厅名字和地址。

餐厅名字的使用很明确，在搜索和为用户显示详情页面时都会用到。地址可以用作位置信息。我们可以利用地理定位服务将餐厅地址转换成相应的经纬度坐标。在搜索时，经纬度地址对于过滤和查找指定范围内的所有餐厅会很有帮助。同时，对于那些和发起搜索请求的用户距离较近的餐厅而言，当我们对其进行放大处理时，经纬度坐标也是很有帮助的。

不只是位置信息，Yowl 的用户还想知道餐厅都提供哪些食物。用户通常会按菜系来搜索餐厅（如墨西哥菜、意大利菜、中国菜等）。有些用户还会用 Yowl 查找某一道菜品。当用户查看搜索结果的时候，如果能看到附近的菜系和菜单信息，以确信其搜索结果是正确的，那么用户会心存感激的。收集这些信息是一件很复杂的事。我们期望用户能够提供对餐厅的评论；或许我们也可以要求他们提供有关这些餐厅缺失的一些详细信息。而且只要我们给更多的餐厅带来规模可观的用户，那些餐厅的老板就会主动地提供这些信息。不过目前，我们或许只能挽起袖子自己来查找和录入这些信息了。



除了位置和菜肴信息，用户还会根据二级条件对搜索结果进行筛选和排序，如：价格、好评率、是否有折扣等。人均消费金额和好评率来自于用户，折扣信息则来自于餐厅。餐厅还可以提供对他们自己的介绍。如果能在详情页面将这些信息呈现给用户，那就太好了，而且这样也会带来丰富的、可供搜索的文本。

最后，不要忘了 Yowl 的成功依赖于会员餐厅。这是一个简单的布尔值，用来控制餐厅是否会得到额外的推荐，使其出现在搜索结果的最前面。类似的，在限定的时间范围内，我们还会对那些积极参与的餐厅做一些推广——它们提供的信息对我们很有帮助。

来看一下，我们需要的信息、获得这些信息的来源，以及信息的使用方法，如表 9.1 所示。

表 9.1 Yowl 的信息需求、来源，及其使用

信息	例子	来源	使用
餐厅名称	Taco Belly	公共数据	搜索，显示（列表、详情）
地址	1234 Cordon Ave. Blacksburg TN 23913	公共数据	地理位置的来源，显示（详情）
位置（纬度 / 经度）	35° 24' 30" N 88° 31' 20" W	根据地址的地理定位计算得到	筛选，显示，距离数据的来源
距离	12 英里	根据用户及餐厅的位置计算得到	放大处理，排序，显示（列表）
菜系	墨西哥餐；快餐	餐厅，顾客	搜索
菜单项	Burrito Ultimate \$4.95	餐厅	搜索，显示（详情），价格信息的来源
价格	\$, \$\$, \$\$\$, \$\$\$\$	菜单项，用户	筛选，显示（列表）
好评率	★, ★★, ★★★, ★★★★★	顾客	筛选，显示（列表）
折扣	（浮动）	餐厅	筛选，显示（列表、详情）
推荐餐厅	布尔值	餐厅	放大处理
填好的表单	布尔值	餐厅	放大处理
描述	“Tasty, Fast, Cheap, Addictive”	餐厅	显示（列表、详情），搜索

## 9.3 搜索应用的设计

当所有要收集的“原材料（ingredients）”都准备完毕后，我们就可以把“食谱（recipe）”放到一起进行餐厅搜索了。如图 9.4 所示，首先，我们会根据预计的用户需求来设计用户体验。接下来，会根据对可用信息的理解找出需要进行索引的字段；这些字段会作为我们的基础信号。我们还需明确要分析处理的具体需求，以确保文

本能得到正确的分词，相应的特征能得到成功的提取。一旦确信这些基础信号能够独立地正常工作，我们就可以构造查询，将这些信号放在一起平衡彼此之间的影响了。

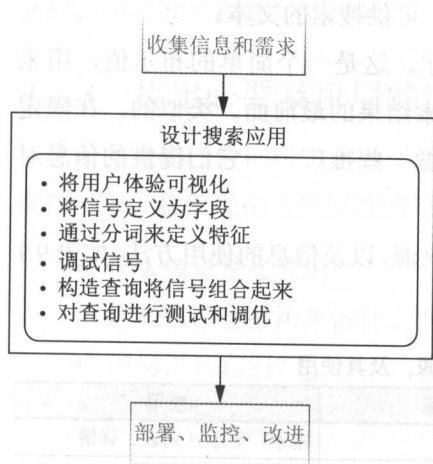


图 9.4 一种设计搜索应用的系统化方法，包括建立恰当的字段和查询，以匹配用户所提的问题。

### 9.3.1 将用户体验可视化

介绍了对用户需求和商业需求的理解，让我们来好好想一想用户体验的问题。从搜索相关性的视角来看，我们有两个目标：首要目标是帮助用户尽快满足他们的信息需求（别忘了，这也是本书的目标）；次要目标是为用户提供相关性反馈（如第8章所述），帮助他们理解为什么会看到当前的搜索结果，以及如何对其做出相应的调整。

当用户打开 Yowl，他们可以通过类似图 9.5 所示的用户界面，立即开始查找餐厅。一些用户是从文本搜索开始的，他们会输入餐厅的名字、菜系或是具体菜品的名字。如果用户没有特别具体的需求，他们就会通过价格和好评度来筛选附近的餐厅。很多用户会同时使用搜索和筛选功能。当用户修改他们的搜索文本或筛选条件时，地图上的标记就会得到更新，这些标记是用来标注满足搜索条件的邻近餐厅的。地图本身可以被用作一个基于位置信息的餐厅过滤器，为了看到不同位置的餐厅，用户可以移动或缩放地图的不同区域。

当我们按用户的搜索条件给出结果以后，系统进而给出了搜索结果的两种视图。第一种视图，即地图视图（map view），是当前浏览界面的一部分。如图 9.6 所示，

我们可以去掉浏览组件以显示地图的全貌。

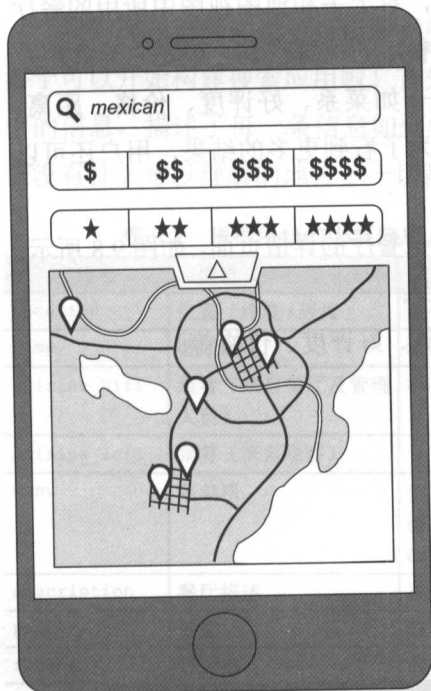


图 9.5 Yowl 的浏览界面

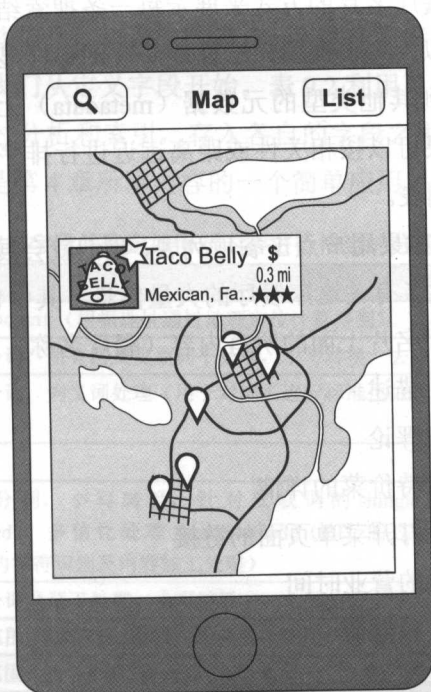


图 9.6 Yowl 的地图视图，搜索结果的两种视图之一。

地图的可视化展现形式为我们提供了相关性反馈的手段，确保了用户能在正确的位置上进行搜索。当用户点击一个代表餐厅的标记时，系统就会弹出一个餐厅的名片，显示的相关信息如下：

- 餐厅名称
- 用户所在位置到餐厅的距离
- 好评度（用星的数量表示）
- 价位（用包含 \$ 符号的个数表示）
- 菜系
- 一幅餐厅的图片

餐厅名片的目的是在屏幕上以尽可能紧凑的方式为用户显示最为重要的信息。这样做可以让用户对视图进行快速浏览，只有当看到有符合条件的餐厅时，他才会通过点击进入详情页面。

如图 9.7 所示, 我们也可以用匹配餐厅列表的形式来显示搜索的结果。这个列表视图以餐厅名片的方式来展示每一条搜索结果, 看上去和地图视图中所用的餐厅名片很类似。列表视图无法像地图视图那样显示餐厅的位置信息, 但是它允许用户将焦点放到其他类型的元数据 (metadata) 上——如菜系、好评度、价格、距离。并且其结果可以按相关性或距离远近进行排序; 为了看到更多的结果, 用户还可以向下滚动列表。

最后, 如果用户点击餐厅列表, 我们就会展现该餐厅的详情页面, 如图 9.8 所示。该页面提供了有关这家餐厅的大量信息, 其中包括:

- 餐厅名片上的所有内容 (餐厅名称、距离、好评度、价格等)
- 餐厅描述
- 用户评论
- 有关特价菜的详情
- 一个打开菜单页面的链接
- 餐厅的营业时间
- 地址和电话

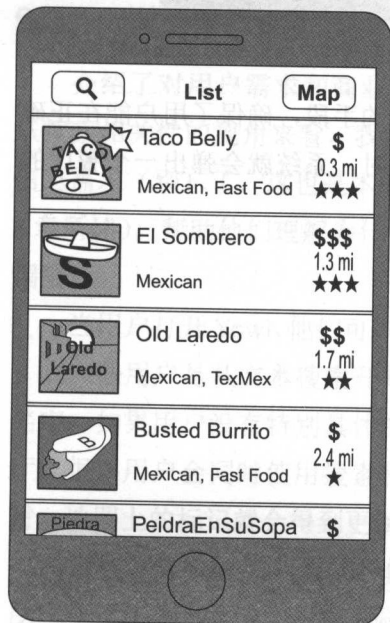


图 9.7 列表视图, 一种以表格形式展现搜索结果的视图。

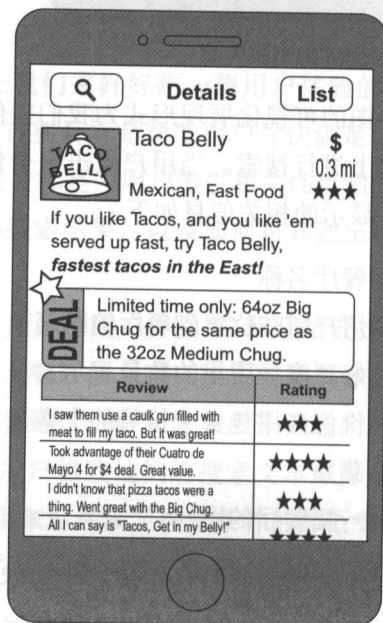


图 9.8 Yowl 的餐厅详情视图

### 9.3.2 定义字段和模型的信号

我们现在已经知道了可以获得哪些信息，以及它们在 Yowl 里是如何被使用的。终于可以开始构建搜索应用啦！首先我们从定义字段开始。表 9.2 利用了来自表 9.1 中的信息，描述了每一条信息如何经过分析和索引，存入各自的字段之中。此处应该没有什么难以理解的地方；一切都是第 4 章所讲内容的一个简单应用。

表 9.2 根据所给的必要信息，定义字段并确定如何对内容进行恰当的分析

字段名	信息（来自表 9.1）	分析
location	位置（纬度 / 经度）	geo_point（根据地址通过地理定位计算得到）
name	餐厅名称	标准分词，所有格及复数词干处理，小写转换，去除重音符号
cuisine_hifi	菜肴（来自餐厅及管理 人员）	标准分词，同义词处理（用于对词汇进行标准化处理）
cuisine_lofi	菜肴（来自顾客）	
menu	菜单项	标准分词，小写转换，针对双联词的 shingle 处理（bigram shingled），多值化处理（multivalued）（ETL 的过程会涉及基于 OCR 的字符识别及内容加工处理）
description	餐厅描述	英语分词及词干处理，小写转换
price	价位	取值范围 { \$, \$\$, \$\$\$, \$\$\$\$ }, 由菜单项或用户评价决定
rating	好评率	取值范围 { ★, ★★, ★★★, ★★★★ }, 由用户评价的均值决定
has_discount	折扣	布尔值
promoted	推荐餐厅	布尔值
engaged	填好的表单	布尔值

虽然我们省略了一些细节，但不要误认为对信号进行建模是一件容易的事情。实际上，对字段的定义有可能需要通过迭代的方式来完成。这是因为经常会遇到意想不到的情况，让我们不得不回到起点，重新对分析进行调整，以应对不时蹦出来的边界情况。

如何以一种自然的方式对基础信号（表 9.2 中的字段）进行分组，行文至此，我们可以开始思考这一问题了。后面，当开始构造查询的时候，我们就可以把这些分组当成逻辑上的整体，而不是把每个基础信号看成完全独立的实体。从现在开始往后的几节，我们将构造出 Yowl 的主体查询。大家会看到，以分组的方式来思考这些字段，可以极大简化平衡信号和调节整体查询的过程。

如我们在表 9.2 中看到的那样，Yowl 的字段可以被划分成相当自然的 4 个分组：位置（location）、内容（content）、用户偏好（user preference）和业务考量（business concern）。其定义如下所示。

- 位置——就是 `location` 字段，一组经过 `geo_point` 编码的经纬度数值对。
- 内容——包含一些文本字段，`name`、`cuisine_hifi`、`cuisine_lofi`、`menu` 和 `description`。注意，我们把菜肴信息分成了两个字段，一个是高保真字段，`cuisine_hifi`，用于表示由 Yowl 的管理人员和餐厅所有者提供的菜肴信息；另一个是低保真字段，`cuisine_lofi`，其所表示的信息来自顾客对餐厅的评价。
- 用户偏好——用户的偏好是由 `price` 和 `rating` 字段来表示的。这两个字段都有相似的 UI/UX（用户界面 / 用户体验），它们允许用户对餐厅进行筛选，在索引中的表现形式也是类似的。两个字段唯一不同的地方就是它们的含义。
- 业务考量——包含了 `has_discount`、`promoted` 和 `engaged` 字段。这些布尔型字段对餐厅的分组进行了定义，此处的餐厅指的是，作为我们商业策略的一部分，得到 Yowl 推广的那一部分餐厅。

### 9.3.3 信号的组合与平衡

到这里，我们可以自信地说，这些字段从其所编码的数据中已经正确地捕获了重要的特征。现在我们可以开始建立查询，对信号加以组合与平衡了，这项任务有时颇有难度。在这一小节中，我们会介绍一种自底向上的方法，将基础信号（字段）根据它们的逻辑进行分组并加以组合。一旦这些位于更高层次的信号构造好后，我们就可以把它们组合在一起，以创建最终用于 Yowl 搜索的查询语句了。

#### 为相关信号构造查询

我们已经看到如何将字段按位置、内容、用户偏好和业务考量进行自然分组。现在让我们来考虑一下，如何对这几个更高级别的信号按照 Elasticsearch 的查询逐一加以描述。

首先，来看一下位置信号。我们需要两种查询来处理位置信息，其中一种是限定框过滤（`bounding-box filter`），它被用于地图视图，在地理位置上对被搜索区域加以限制，请看下面的清单 9.1。



### 清单 9.1 限定框地理过滤器

```
{  "filter": {
    "geo_bounding_box": {
      "location": {
        "top_left": <northwest corner of user display>,
        "bottom_right": <southeast corner of user display>}}}}
```

另一种位置查询则根据餐厅与用户当前位置在地理上的距离进行评价, 如下面的清单 9.2 所示。

### 清单 9.2 基于地理信息的查询, 距离用户越近的餐厅评价越高

```
{  "query": {
    "function_score": {
      "functions": [{
        "gauss": {
          "location": {
            "origin": {
              "lat": <user-lat>,
              "lon": <user-lon>,
              "offset": "0km",
              "scale": "10km"}}}}}}}}
```

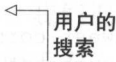
这条查询确保了在 Yowl 的列表视图中, 附近的餐厅相比于距离较远的餐厅, 其排名会更加靠前。由于根据距离来决定评价在计算成本上可能会比较高, 因此每次使用这一查询时, 我们也会使用清单 9.1 中的地理过滤器。否则, 我们就不可避免地会对那些用户并不感兴趣且距离较远的餐厅进行评价计算了。

当构造相应信号的子查询时, 如我们在第 7 章中所做的那样, 单独测试每一条查询也是很重要的。当我们要把这些内容组合到一起形成一条完整的查询时, 这种做法可以简化所需的工作, 因为那样我们就可以把关注点放在更高级别的信号上, 而不必担心这些基础信号之间有没有得到适当的平衡。通过对一组真实的文档进行索引, 然后执行每一条查询, 确保其能够有效度量期望的信息, 我们用这样的方法来测试前述查询。在本例中, 我们可以验证返回结果中是否只包含指定范围内的餐厅, 并且按照距离从近到远排序, 以这样的方式来测试位置过滤器 (location filter) 和查询语句是否正常工作。

接下来, 我们要构造一条处理“内容”信号的查询。这是我们基于文本最为主要的一条查询, 参见清单 9.3。

## 清单 9.3 与内容有关的查询

```
{ "query": {  
  "multi_match": {  
    "query": <user query>,  
    "fields": [  
      "name^10",  
      "cuisine_hifi^10",  
      "cuisine_lofi^4",  
      "menu^2",  
      "description^1"],  
    "tie_breaker": 0.3}}}
```



正如我们在前面几章详细讨论过的那样，搜索文本字段有许多策略可以使用：查询可以以字段为中心（field-centric），也可以以词为中心（term-centric），评价可以是对子评价的简单求和，也可以是更为复杂的算法，比如带 tie 参数的 disjunction-maximum 评价。或者我们也可以抛开所有复杂的处理手段，仅仅把文本导入一个单独的内容字段里，只考虑对这一字段的单一评价。在前面的例子中，我们使用的是一个以字段为中心的方法，它利用了 multi\_match 查询，涉及字段 name、cuisine\_hifi、cuisine\_lofi、menu 和 description。默认情况下，multi\_match 查询使用的是 best\_fields 方法，这种方法在这里很合适，因为用户可能会搜索餐厅名称、菜系类别，或菜单上的一道菜，但不会三个同时搜索。不过为了缓和 best\_fields 方法的行为所带来的影响，我们还要引入一个 tie\_breaker 参数，为其取值 0.3。

清单 9.3 中所列的字段都是跟放大处理有关的，放大的倍数反映了每一个字段所处的重要级别。我们最好能认真思考一下这些放大处理，为每个字段设置一个代表放大倍数的值。这些放大值会根据我们对相关性的微调而相应地有所改变。由于典型的用户使用场景会涉及用户对餐厅名称或菜系类别的搜索，因此这两个字段得到的放大值是最大的。回忆一下之前所说的，cuisine\_lofi 的内容是由顾客提供的，因此相比而言我们会更加相信 cuisine\_hifi 的内容。基于这一原因，我们给 cuisine\_lofi 赋予了较低的权重。menu 字段对搜索某道菜品的用户来说非常重要，但我们却给了它一个较低的权重，这是因为我们不希望碰巧出现在 menu 字段中的菜系类别，其权重会胜过 cuisine\_lofi 和 cuisine\_hifi 字段中的匹配。最后，我们给 description 字段设置的放大值是最小的，这是因为它的目的只是为了对内容搜索加以补充。

通过确保只有预期的文档才能得到匹配，我们可以对内容查询是否正常工作加

以测试。通过确保文档按正确的顺序得到返回，我们可以对评价功能是否正常工作加以测试。但这样的测试有一些主观。很快，我们会学到一种测试驱动的相关性技术（test-driven relevance）——这是一种能够让测试尽可能客观的技术。

接下来是代表用户偏好的信号，这是我们的搜索中最为简单的部分了。当用户选择价位（也就是 \$ 符号的数量）或好评度（也就是星的数量）时，他们是想过滤掉所有其他的结果。因此，这里不涉及评价计算。我们只要一个过滤器就可以了，如清单 9.4 所示。

#### 清单 9.4 按价位或好评度来约束匹配结果的用户偏好过滤器

```
{
  "query": {
    "bool": {
      "filter": [
        {"match": {
          price: "D"}},
        {"match": {
          rating: "S"}}]}
  }
```

此处我们使用 D 来表示单个 \$ 符号，S 表示星。这个过滤器很容易测试；只要确保搜索结果只包含满足指定价位和好评度的餐厅就可以了。

最后，我们来考虑一下商业需求。目标是要对某些餐厅进行放大处理，比如作为服务会员的那些推荐餐厅，通过我们的服务来打折促销的餐厅，或者是积极参与内容提供的餐厅（它们为我们提供了很有价值的信息），参见清单 9.5。

#### 清单 9.5 考虑了商业因素的查询，对付费和积极参与的客户进行推广

```
{
  "query": {
    "function_score": {
      "functions": [
        "filter": {
          "bool": {
            "should": [
              { "term": { has_discount: True }},
              { "term": { promoted: True }},
              { "term": { engaged: True }}}]},
          "script_score": {
            "script": """
              0.5*doc["promoted"].value +
              0.3*doc["has_discount"].value +
              0.2*doc["engaged"].value
            """}}]}
  }
```

简单起见，我们将评价逻辑写入了一段评价脚本中，给接受推广的餐厅赋予了最大的放大值，紧接着是那些提供折扣的餐厅，再往后则是那些没有付费但是积极

参与内容提供的餐厅。评价脚本的计算成本有可能会很高，因此，为了限制待处理餐厅的数量，我们需要对数据进行筛选，只包含一部分餐厅，且至少满足如下几个条件之一：接受推广的、有折扣的，或是积极参与的。要注意的是，这样做不会过滤掉任何搜索结果；它只是对接受 script\_score 处理的文档进行了限定。对这条查询的测试也很简单：创建一个文档集合，包含上述三类条件的每一种组合，并确保每一篇文档都得到了期望的评价。

### 子查询的组合

至此，已经在我们的搜索应用中为每一个主要信号构造了相应的子查询。在构造每一条子查询的时候，我们也对其进行了测试，以确保文档匹配无误，结果评价得当，从而使查询结果能够正确反映我们的相关性目标。在这一节中，我们将通过构造一条支持 Yowl 搜索的总查询，在抽象层面上再上升一个高度，参见清单 9.6。

清单 9.6 合并了所有主要信号的组合查询

```
{
  "filter": {
    "bool": {
      "filter": [
        {
          "geo_bounding_box": {
            location: { <user's bounding box> } }},
        {
          "match": {
            "price": <user's price preference> }},
        {
          "match": {
            "rating": <user's rating preference> } } ] },
    "query": {
      "function_score": {
        "score_mode": "sum",
        "query": {
          "multi_match": {
            "query": <user's search terms>,
            "fields": [
              "name^10",
              "cuisine_hifi^10",
              "cuisine_lofi^4",
              "menu^2",
              "description^1" ],
            "tie_breaker": 0.3 } },
          "functions": [
            {
              "weight": 1.2,
              "filter": {
                "bool": {
                  "should": [
                    { "term": { has_discount: True } },
```

位置过滤器

用户偏好

① 对相关性评价值求和

内容

业务因素

```

    { "term": { promoted: True }},
    { "term": { engaged: True }}}}},
"script_score" : {
  "script": """
    0.3*doc["has_discount"].value +
    0.5*doc["promoted"].value +
    0.2*doc["engaged"].value
    """},
{
  "weight": 2.1,
  "gauss": {
    "location": {
      "origin": { <user's location> },
      "offset": "0km",
      "scale": "4km", }}}},
{
  "weight": 1.0}]}}}}

```

位置

神秘的内容权重

这条查询的最外层由两部分组成，分别是 `filter` 和 `query`。其中，`filter` 限定了结果集中只包含位于用户当前位置附近的餐厅，并满足用户对价格与好评度的要求（如果指定的话）。而 `query` 部分则更为复杂一些。其中，第一部分是内容查询，在整个查询的上下文中，内容查询是用来对餐厅做进一步筛选的，即只返回包含用户搜索词的那部分餐厅。但更重要的是，该查询为每一篇文档提供了一个基本的相关性评价；而其他信号（位置因素和业务因素）则只用来在该评价的基础上进行一定程度的微调（放大）。在 `query` 这一段余下的部分中包含了一系列函数（位于 `function_score` 查询中），它们提供了前述的这些放大功能。每一个函数都包含了一个权重参数，允许我们对信号加以平衡。一共有三种权重：一个是对位置的权重，一个是对业务考量的权重，另一个则是对内容的权重。用户偏好没有权重，因为它们只是用来对内容进行筛选的。

至此，大家可以清晰地看到，我们一开始就为每一个高阶信号创建子查询的重要性了。在之前的查询里，我们只有三个“旋钮”可以调节，以此来控制相关性行为——也就是前述的三个权重。如果不是通过构建子查询入手，那就得同时考虑所有 11 个字段之间的相互作用，在这种情况下，要调试一个相关性问题的几乎是不可能的。

### 理解信号权重的行为

也许大家已经注意到了，在清单 9.6 中有些地方比较奇怪：内容子查询和与之对应的权重是分开的。为了更好地理解每一个权重的行为，让我们再仔细看一看查

询的结构。在本节的最后，我们还会回过头来概括一下我们所学的内容，以便大家可以吧同样的思路应用到自己的搜索项目中去。

在 Elasticsearch 中，一个 `function_score` 查询包含了 `query` 部分和 `function` 部分。其中，`query` 部分没有什么特别的，它是用来对文档进行匹配和评价的。值得注意的是，`query` 中的评价是如何与 `function` 中产生的数值相互作用的呢？默认情况下，每个函数产生的数值结果都是彼此相乘的，而最终的结果则会与主查询的评价值相乘。用等式来表示，如下所示：

$$\text{TotalScore} = (\text{BusinessScore} \times \text{LocationScore}) \times \text{ContentScore}$$

但是因为这里包含了 `'score_mode': 'sum'`（如清单 9.6 中 ❶ 所示），其行为也就发生了改变，在最终与 `query` 的评价值相乘之前，所有函数的取值会先被求和累加。如清单 9.7 所示。

#### 清单 9.7 表示总体评价计算的等式

$$\text{TotalScore} = (w_B \times \text{BusinessScore} + w_L \times \text{LocationScore} + w_C) \times \text{ContentScore}$$

此处我们用到的权重（ $w_B$ 、 $w_L$  和  $w_C$ ）与清单 9.6 中的权重相对应。它们分别对应于业务、位置和内容的权重。在这一抽象层次上，我们不必直接对 `BusinessScore`、`LocationScore` 和 `ContentScore` 的值进行控制，这些值都是由之前用于构造主查询的那些子查询来决定的。但是通过调整  $w_B$ 、 $w_L$  和  $w_C$  的权重，我们依然可以仔细平衡 `BusinessScore`、`LocationScore` 和 `ContentScore` 之间的相互作用，从而控制搜索的相关性。

要注意，在前述等式中的那些信号是不对称的。`ContentScore` 会分别与 `BusinessScore` 和 `LocationScore` 相乘。在一定程度上，这使得针对内容的评价与其他信号相比更有优势；如果内容的评价较低，那么要想抬高总体评价就会比较困难。另一方面，如果我们把针对内容的评价计算改成累加求和，可以让等式变得更加对称：

$$\text{TotalScore} = w_B \times \text{BusinessScore} + w_L \times \text{LocationScore} + w_C \times \text{ContentScore}$$

这样做就可能会出现文档评价很高，但是内容评价却为零的情况！这显然会带来非常糟糕的用户体验。用户可能会搜索“sushi”，而 Yowl 应用没准会推荐一家隔壁的汉堡连锁酒店！为了避免这种意外情况的发生，把对内容的评价和其他因素的评价相乘是最好的选择。



那么, 清单 9.6 中那个针对内容的神秘权重是什么呢? 为什么这个权重是必要的呢? 为了彻底回答这一问题, 让我们考虑一下如果没有针对内容的权重, 搜索会如何执行呢? 用等式来表示的话, 这有点像清单 9.7 中的样子, 只是少了  $w_c$ :

$$\text{TotalScore} = (w_B \times \text{BusinessScore} + w_L \times \text{LocationScore}) \times \text{ContentScore}$$

记住这一等式, 让我们来假设一种糟糕的情况。试想有这样一家餐厅, 它很好地匹配了用户的查询, 但就是没有商业价值 (也就是没有被推广、没有参与内容提供, 或给予任何折扣)。另外, 这家餐厅的位置坐落在指定地理位置范围的边缘, 其位置评价因此也非常低。如果我们使用这个不带内容权重的等式, 那么尽管内容上绝对匹配, 但这家餐厅的整体评价将会很低。实际上, 我们的业务评价和位置评价是先结合在一起然后再与内容评价产生关联的。但是, 如果我们像清单 9.7 中所示的那样, 把内容权重加进来, 就可以对“业务评价和位置评价先结合再与内容评价产生关联”这一行为所带来的影响加以控制了。这样一来, 当  $w_B$ 、 $w_L$  和  $w_c$  被适当调整之后, 搜索结果就会把关注点适当放在内容的质量上, 而不会过度依赖于相对没有那么重要的位置和业务信号了。

让我们回过头来总结一下前面的讨论。当开发自己的搜索应用时, 我们所构造的查询, 其结构可能会类似于清单 9.6 中的 Yowl 查询。这样的查询由一个 `function_score` 查询组成, 里面有一个 `query` 子句, 会结合内容对文档进行匹配和评价, 此外还有一个函数, 会承担额外的放大逻辑。不过, 即便我们有再多千奇百怪的查询, 有一点是很重要的, 那就是对查询结构的理解, 就像我们现在理解 Yowl 查询的结构一样。我们要能够写出一个等式, 可以解释如何将高级别信号组合起来, 构造出整体的评价。还要考虑那些极端情况, 确保用户不会遇到由于次要信号压制重要信号而导致的不正确的行为。

### 搜索的整体调优和测试

有了结构合理的查询之后, 我们还有最后一项任务要做, 那就是: 调整权重, 以适当平衡内容、用户和业务这三种信号之间的关系。不过, 这项任务虽然听上去很容易, 可在所有和相关性有关的工作中, 它却常常是最容易让人受挫和耗时的。在这一节中, 我们会介绍一种对查询参数进行调优的结构化方法。

首先, 我们会把焦点放在基于内容的相关性上, 而忽略其他因素, 如: 用户偏好或业务需求。在对基于内容的搜索做了优化之后, 我们再来看用户偏好。最后,

在用户体验达到最佳效果之后，我们再把业务需求引入查询之中。这一过程完成以后，我们获得的搜索体验就会变成：在兼顾实际业务需求的前提下，优先考虑用户的需求。还是那句话，虽然我们讨论的是 Yowl 的查询，但这里介绍的原理也适用于我们自己的搜索项目。

在开始调节查询参数之前，为了进行测试，我们首先需要对一组真实的文档进行索引，而且数据集越大越好。理想情况下，可以选取产品索引的一个快照，并用它来调试查询参数。我们还需要一组具有代表性的搜索请求。对于一个已存在的应用而言，我们可以通过扫描搜索日志来寻找典型的搜索请求。

因为是第一次构造 Yowl 搜索，所以我们需要发挥一点想象力，推测一下用户可能会问到的问题类型。关键的一点是，我们需要寻找一些用户请求的例子，能够针对各种典型用例进行练习。由于我们之前已经为用户创建了角色代表，因此参考一下他们所提的问题，确保至少能够涵盖我们期望看到的全部用例，这样做也许不失为一个好办法。

首先我们要调节的是针对内容的子搜索。还记得它吗？

```
{ "query": {  
  "multi_match": {  
    "query": <user query>,  
    "fields": [  
      "name^10",  
      "cuisine_hifi^10",  
      "cuisine_lofi^4",  
      "menu^2",  
      "description^1"],  
    "tie_breaker": 0.3}}}
```

此处，我们有 6 个参数可以调节，其中一部分参数是针对各个内容字段的，剩下一个则是针对 tie\_breaker 的。我们可以通过如下几个步骤来对这些参数进行调节：执行典型的搜索请求；查看每个请求的返回结果；通过修改参数来解决我们遇到的任何问题。正如我们前面所提到的，通常这样一个过程是比较主观的。在下一节中，我们会介绍一种测试驱动的相关性方法（test-driven relevance），它可以使这一过程相比于典型的、“仅靠尝试大量查询然后观察执行结果”这样的方法显得更有条理，也更加客观。

当以这样的方式调节文本查询时，我们会不可避免地遇到一些难以解释的结果。当这种情况发生的时候，记得要借助 Elasticsearch 内置的查询解释功能（通过在查

询中添加 'explain': true 来启用)。正如我们在第 3 章中讨论的那样,解释信息的细节一开始读起来有点困难,但是对于理解为什么文档可以匹配某个特定的查询,以及它们如何进行评价这样的问题,这些解释信息是非常有用的。

在对内容子查询的结果感到满意之后,我们再把焦点转向完整的 Yowl 查询,如清单 9.6 所示(当然,我们会根据之前得到的匹配值来更新内容子查询中的参数)。在此基础上,我们要做的是为三个权重找到合适的值:内容权重  $w_c$ 、位置权重  $w_L$  和业务权重  $w_b$ 。让我们先把  $w_c$  设为 1.0,把  $w_L$  和  $w_b$  设为 0.0,从这一点出发,以此为基准,文档只根据其内容进行评价,位置和业务因素对总体评价都不会产生影响。(参考清单 9.7 中的等式,大家能看出为什么是这样的吗?)

因为要处理完整的查询,所以我们需要再次使用位置、价格和好评度过滤器。紧接着,通过将反映典型搜索请求的位置过滤器包含进来,我们希望这个搜索示例能够更加真实一些。可是,马上我们就会发现,基于内容的相关性出现了下降的情况。大家能想到这是为什么吗?下降的原因是由于查询到的文档数量变少而导致的。当我们在整个索引范围内查找时,可以匹配到的餐厅很多,但是当把焦点集中到一个特定的区域时,对于一条给定的查询,就可能不会有太多相关性结果返回了。不必担心,至少我们知道我们的搜索正越来越接近真实的状况。如果这样的搜索结果很糟糕,那么再怎样对查询调优也是没多大用处的;我们只需要在索引中加入更多的餐厅数据就可以了——而这这是一个商业问题,并非技术问题!

接下来,让我们再看一下位置的影响。为此,我们需要逐步增大位置权重  $w_L$  的值,同时注意搜索结果在质量上的变化。当值为 0.0 的时候,位置查询对搜索结果没有什么影响。随着  $w_L$  取值的增加,我们注意到文档的排列顺序会重新调整,那些距离较近的餐厅会比距离较远的餐厅排名更加靠前。如果  $w_L$  的值增加太多,我们就会看到,一些不太相关的结果会跑到前面去。

一旦我们找到合适的  $w_L$  值,内容信号和位置信号就取得了平衡。到这里,我们已经为我们的用户构建出了一个接近最优的搜索应用。但遗憾的是,我们自己的业务需求还没有任何一条得到满足!因此,是时候来看一下业务信号了。比较直接的办法,就是从 0.0 开始慢慢地增加业务权重  $w_b$  的值,直到看到相关性下降为止。但是刚才我们已经把位置权重的值加得很大了,任何来自业务信号的进一步影响都可能会导致相关性的迅速下降。

我们需要一种不同的方法,一种以对称的方式来处理位置信号和业务信号的方法。

法。为了达到这一效果，一开始我们通过把  $w_L$  的值设为 0.0，将位置信号临时关闭。然后再使用与之前相同的技术，增加  $w_B$  的值，直到相关性开始明显受到影响为止。一旦找到合适的  $w_B$  之后，我们就需要将位置权重  $w_L$  恢复到之前已确定的值。因为我们已经以对称的方式对位置信号和业务信号进行了处理，现在它们两个彼此之间应该有了相当好的平衡。不过由于现在同时有了来自业务和位置的影响，内容信号就有可能会受到抑制。虽然我们有多个“旋钮”需要控制，但所幸的是，这不是什么难题。通过逐渐增加内容权重  $w_C$  的值，直到相关性得到恢复，我们能够在内容、位置和业务这三个因素之间达到一个最佳的平衡。

遗憾的是，在真实的搜索应用中，对参数的调节可能要比这种理想化的例子更加复杂。但是，为了避免不必要的复杂性（例如，尝试对所有字段进行独立调节），至少我们要以这样的理想情况为目标，这一点是很重要的。另外，即使调节过程变复杂了，之前所讲的思路也依然是适用的：我们要理解信号之间相互作用的规律，以便合理地对参数进行调整。

### 是啊……可是我们如何对相关性参数进行调节呢

或许大家已经注意到了，在我们完成的所有步骤中，很多时间都花在了对参数值的调整上面。并且每一步我们都说了，在开始下一步之前，要确保查询结果“看着还行”。但是，我们并没有讨论过，如何才能确保参数值是朝着正确的方向在调节。

对于初学者来说，我们的这条最优的评判标准“看着还行”，其说法是比较含糊的。为了让搜索结果达到最优，我们必须理解用户所认为的相关性指的是什么。但这一点是很难的。例如一位用户搜索“taco”，其目的是为了搜索一个名称里带有“taco”的餐厅（也许是用户忘了餐厅的全名），而另一位用户搜索“taco”，其目的则可能是为了搜索提供 tacos（玉米饼）的餐厅。因此，我们的最佳相关性这一概念，必须对用户不同的信息需求有所了解才行。此外，“taco”只是一条查询而已；我们必须确保，对用户可能发起的任何查询都要保持相关性。如果每位用户搜的都是“taco”，我们可以简单地对参数进行调整，直到对玉米饼的搜索看起来完美无缺为止。相反，如果还有 100,000 条其他类型的搜索，也需要“看着还行”，那么只是过分关注这其中的任何一条，对于其他查询来说肯定都是不利的。

鉴于此，我们应该怎样做才能保证对参数的调整不会误入歧途呢？首先，我们需要有一组比较有代表性的查询。它应该包括那些普通的查询，还应该包括我们能够想象得到的、各种用户使用场景下的查询，甚至应该包括一些随机的查询——理

想情况下，就是从查询日志文件中搜集到的各种查询请求。

现在，对于一组给定的典型查询来说，我们需要搞明白，“看着还行”对其中每一条查询都意味着什么。这句话可能和“我只要看到它，就知道是怎么回事了”一样的含糊，但在理想情况下，我们应该对其有更为具体的定义才行。说得极端一点，在信息检索领域，人们是利用所谓的相关性判断（relevance judgments）来对“看着还行”这样的说法加以定义的。这里，给定一组固定的查询和一组固定的文档，每一对“查询 - 文档”按照文档与相应查询的匹配程度被分成 5 个等级。我们将这些经过分级的“查询 - 文档”配对笼统地称为判断（judgment）。

手里有了一组查询，对于“还行”也有了一个较为合理的认识，我们终于可以开始对参数进行调节了。这一过程涉及：发起一系列查询，同时查看每条查询的返回结果，然后在出现相关性问题时，对其进行识别和诊断。如果我们对这些查询如何工作有很好的理解，那就应该对如何修改搜索参数也会有很好的办法。比如在之前 Yowl 的例子中，如果搜索结果的第一页中只包含了匹配查询相对较弱的临近餐厅，那么位置信号就应该被赋予较低的权重。

不用说，收集相关性判断是一个体力活。如果我们发现自己经常要不断检查相关性设置，那么花些时间来自动化这一过程也许是值得的。我们可以构造一个相对简单的应用，允许内容管理员发起查询，查看搜索结果，并针对给定的搜索条件来标记结果正确与否。当我们有足够的查询和文档被标注（有足够的关联性判断）时，就可以实现相关性测试的自动化了。这样一来，每次我们修改搜索参数时，所有测试查询都会被重新执行，新的设置将根据全部查询测试的执行效果来进行打分。

我们称这种自动化的方法为测试驱动的相关性（test-driven relevance）。利用这种方法，我们再也不用手动执行搜索了，更为重要的是，我们再也不用记住所有搜索中每条搜索的行为了。像 Quepid (<http://quepid.com>) 这样的工具，还可以帮助我们简化这一流程。

最后，参数调整的理想解决方案是将自动化过程再往前推进一步。只要有足够的用户操作，许多技术都可以被用来自动收集相关性判断。有了这些相关性判断，程序就可以替我们自动调整查询参数了。在信息检索领域，最高级别的自动化被称为排序学习（learning to rank）。排序学习是一个处理起来较为复杂的问题，但它已经成为人们在信息检索领域的一个关注点。因此，大家不妨留意一下将来人们在这一技术上的改进和突破。



在第10章中，我们对排序学习和测试驱动的相关性方法还会有更多的介绍。

## 9.4 部署、监控和改进

最后，经过我们对查询参数的一番微调，Yowl 餐厅搜索应用的上线已经准备就绪了！走到这一步，我们已经做了很多工作，涉及搜索应用的规划、设计和实现。我们可以松一口气了，因为最难的那部分工作已经在我们身后了。但是，从许多方面来看，部署搜索应用才意味着我们真正工作的开始。现在，我们进入了监控、更新，和以迭代方式改进搜索应用的这样一个循环之中，如图9.9所示。

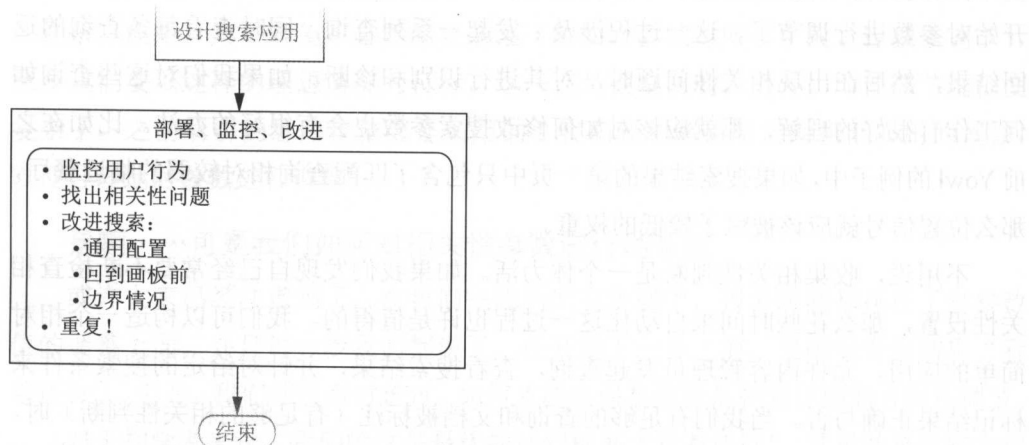


图9.9 搜索应用的部署是一个循环的开始，还涉及监控和以迭代方式改进搜索。

### 9.4.1 监控

在搜索应用的设计过程中，许多选择都是主观的——从用户界面的外观和体验，一直到查询参数的取值，我们都是依靠直觉来做出决定的。不过一旦应用上了线，我们就会得到一种新的信息来源，可以反映搜索应用的质量。这种新的信息来源是什么呢？那就是用户行为。

如果用户理解 Yowl、乐于使用，并能从中获益，这些在用户行为上是会有所体现的。如果用户在访问应用时遇到了问题，我们可以通过分析用户的行为来跟踪并纠正这些问题。同样的，如果我们所做的改动引入了新的问题，那也应该能够看到用户行为的相应变化。因此，从部署应用程序的那一刻起，收集用户信息这一点就



尤为重要。用户的行为信息有许多潜在的来源，但在这里我们只列出了其中一部分，目的是为了对可能的来源给大家一些启示。

- 页面停留时间 (*time on page*) ——如果用户只是匆匆看了一眼搜索结果就迅速离开了，这说明他们并没有找到想要的内容。另一方面，如果我们的相关性反馈非常好，用户也可能会很快找到答案，然后离开。
- 点击率 (*click-through rate*) ——当用户正在通过点击行为查找更多细节信息时，我们就知道他可能已经找到了有价值的东西。
- 转化率 (*conversion rate*) ——如果用户购买了产品（或者在 Yowl 的例子中，比如抢到了一家餐厅的折扣），这说明搜索是起作用的。这一点是最起码的。
- 用户黏度 (*retention*) ——如果用户会定期访问应用，就说明他们觉得这个应用是有用的。
- 深入翻页 (*deep paging*) ——如果用户经常需要翻到搜索结果的第二页或第三页，这也许就是相关性问题的一个征兆。为什么结果不在第一页呢？
- “弹簧单高跳”现象 (*pogo-sticking*) ——用户从搜索结果的列表中点击其中一项，但很快发现这不是他们想要的内容，于是又回到了搜索列表页面。这一行为揭示了相关性反馈存在问题。理想情况下，用户应该在点击进入详细页面之前对文档是否值得查看有一个准确的判断。
- 抖动或反复调整 (*thrashing or reformatting*) ——用户因为没有找到满足其信息需求的内容而连续多次改变搜索条件。这可能说明用户有一个复杂的信息需求，而我们的相关性反馈为其指明的方向是正确的。

无论我们选择对什么样的指标进行跟踪，随时间而变化的相对值往往要比绝对值更为重要。例如，某个电商网站的“深入翻页”现象可能表明了，用户没有找到其所需的内容；但对于法律或专利搜索而言，“深入翻页”可能表明，用户已经发现了大量有用的文档。由于我们要经常检查用户行为的变化，因此，收集并跟踪用户行为指标的基准值 (*baseline values*) 是很重要的。有了基准值，一旦发现指标在数值上有任何突变或下降的情况，我们就可以进一步对问题展开调研。

搜索日志是另一个值得关注的信息来源。通过查看用户发出的搜索请求，我们对用户及其行为习惯的理解会变得更加清晰。

## 9.4.2 找出问题并解决它们

在跟踪搜索历史 and 用户行为的时候，随着数据量的不断增长，一些搜索问题也越发容易被暴露出来。其中最容易发现的问题是“零返回结果”的搜索（zero-results search）。相比于“对不起，没有找到您想要的结果”这样的返回结果，我们几乎总可以找到更好的解决办法。而且，发现“零返回结果”的问题和使用 `grep` 来查询搜索日志一样，都是非常简单的。

更通俗一点说，当看到某个指标有意外下降的情况时，不妨试着找一找其背后的“模式”（pattern）。通常情况下，经过仔细调优的搜索应用还出现问题，往往反映了相关性的某些边界情况。我们能从用户所采取的策略（user strategies）中找到固定的模式吗？也许 Yowl 的用户按照菜系搜索可以找到他们要找的餐厅，而用某一道特定的菜品来搜索却有可能一无所获。

这背后的问题或许与某个特定的主题（topic）或某一组查询关键字的表现不佳有关。例如，随着用户逐渐开始使用 Yowl，我们注意到那些寻找法国菜的用户，其用户体验似乎变得特别糟糕。在众多搜索“French food”“French cooking”和类似关键字的用户中，我们看到了大量的抖动、点击率低下，以及用户黏度不够的情况。通过查看搜索日志，并亲自尝试了某些有问题的查询，用户体验不佳的原因浮出了水面：原来，包含关键字 `french` 的查询，其每次返回的结果，除了快餐厅之外再也没有其他内容了！于是，我们把 `explain` 的值设为 `true`，仔细研究排名过程，很快就意识到了 `French fries`（即，炸薯条）是问题的根源。找到了相关性问题的根本原因之后，我们就可以着手进行适当修正了。

找到了相关性的问题，我们就可能需要在搜索应用的各个层面采取相应的措施来解决问题。以下是各个层面的一些例子。

- 配置——索引通常是分散在多台服务器上的，因此每台服务器上只保存了全部文档的一部分。由于文件数量在每个服务器分片（shard）上都是不同的，有时这会对相关性带来负面的影响。更改索引配置可以解决这个问题。
- 文本分析——在之前有关 `french fries` 的例子中，我们可能需要在同义词文件中加上一项，以确保 `french fries` 不会和法国菜混为一谈。另一种常见的解决办法是将某些词加入受保护单词（protected words）的文件中，以便其不会被提取成错误的词根。

- 信号建模——有些信号对相关性搜索而言可能是不必要或有害的。例如，当我们观察 Yowl 的用户时，可能会发现 `description` 字段的内容太过杂乱而用处不大。在生产环境中经过测试之后，我们可以选择降低 `description` 字段的放大值或者将它完全去除。另一种情况是，有时我们可能会漏掉某些重要的信号，这些信号需要被纳入搜索之中。
- 查询——当用户需求 and 业务需求有所改变时，为了体现这些改变，我们需要在查询中添加或删除一部分内容。如果 Yowl 决定采用分级服务，我们也许会对高级会员进行额外的放大处理。为此，我们可能引入一个新的 `function_score` 函数，并重新平衡构成查询的各个因素。
- UI 上的相关性反馈——我们需要观察用户与应用之间的交互，并确保用户理解其所看到的结果。用户应该能够很容易地找到要找的内容，并理解所呈现的结果。如果不是这样的话，那就应该通过改进我们所提供的相关性反馈来解决这一问题。例如，Yowl 可以为用户提供关键词自动完成功能来引导他们得到有效的搜索结果。

每当我们成功地解决了一个棘手的相关性问题，别忘了还有更多的问题等着我们去发现。搜索应用的维护是一个需要持续保持警惕的过程。除了尚未发现的问题之外，文档库中的每一篇新的文档、每一条新的查询以及每一种新的业务需求都有可能潜在地引入新的相关性问题。但请别担心！监控用户行为指标、发现和修复问题，并测试解决方案，这样一种井然有序的方法会确保我们的搜索应用能够正常运行，并让用户满意。

## 9.5 知道什么是恰到好处

和构建一个强大的搜索应用同样重要的是，我们要知道什么时候应该点到为止（见图 9.10）！在构建搜索应用时，我们其实是在为自然语言进行建模。但自然语言不是一种“纯净”的东西。例如，英语中就有大量的单词，它们在不同情形下会有不同的含义。一些英语单词甚至会充当自身的反义词！例如，想想 `cleave` 这个单词，它有分离的意思，也有紧紧黏着的意思。看到了吧！英语是不“纯净的”！

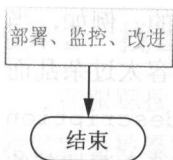


图 9.10 构建一个高度相关的搜索应用很重要，但知道什么时候点到为止也很重要。

因此，别忘了这个绝对的真理：

在构建和完善搜索应用时，我们终将会遇到边际效益递减的规律（the law of diminishing returns）。

在开发 Yowl 搜索的基本功能时，当我们开启搜索引擎并导入餐厅数据时，大概会有 50% 的可能性是奔着“完美”的搜索而去的。通过对各个字段的特征进行更为细致的考量，以及对查询的构造与调优，我们可以让这个数字跃升到 85%。等部署了搜索应用，找出并解决了显著的相关性问题之后，我们就可以让这个数字提升到 93%。要想超越这个值，我们需要投入大量的工作，并且可能只会得到一点点增量收益。而且任何时候，我们把搜索配置得越“完美”，就有可能让它变得越脆弱，越难以改变。最终，我们可能会达到这样一个状态，按照我们在某一刻对搜索应用的狭义理解，我们实际上是在对搜索应用进行过度拟合（overfitting）。任何与这一理解相矛盾的新增文档或业务需求都会导致新的相关问题出现。

为了防止对搜索的过度拟合，请退一步想想，并问问自己：“这么做值吗？”如果我们的改动保证能够修复一个重要的边界问题，那么就放手去做吧。但如果搜索已经表现得不错了，而我们只是想提升一点点性能，那就要小心了——无意中我们可能会让搜索变得脆弱和难以改变。

## 9.6 本章小结

- 开发一个搜索应用的典型步骤，包括：利用角色代表对用户进行建模，识别用户的典型需求，以及设计一个满足需求的应用。
- 对新的信息来源进行整合，同时支持用户和商业两个方面的相关性需求。
- 通过转换（transformation）与分析（analysis）的处理手段来对数据源进行转换和清理，这样做可以实现较低级别的用户和业务匹配规则。
- 低级别信号可以被分成更高级别的信号，它们可以用来对用户和业务在排

名上的不同关注点加以平衡与组合。

- 搜索应用一旦部署成功，就应该根据用户交互来加以监控，以此找出新的问题和意料之外的系统用例。
- 在解决相关性问题的边界情况时，请记住边际效益递减的本质。记住这一点，有助于防止我们根据某一时刻对需求的狭隘理解，而产生的对搜索过度拟合的情况。

# 10 以相关性为核心的企业

## 本章要点

- 改进组织机构在相关性问题上的工作方式
- 解释用户行为数据，以评估搜索的相关性
- 让更广泛的群体意识到搜索的相关性及其价值
- 与领域专家、市场和其他同事开展合作
- 将内容管理员和信息科学家纳入搜索团队
- 利用测试驱动的相关性技术来度量搜索的正确性

到目前为止，我们讨论了如何满足搜索相关性在技术方面的要求。我们还没有讨论过满足这些要求背后的根本原因是什么。如果你是一名软件开发人员，你就应该知道，在一项功能或一个产品上连续工作几个月是很容易发生的一件事情。只有当我们最终将其发布，才会发现市场对我们的工作成果并没有多大兴趣。我们的辛勤付出都白白浪费了，因为对于我们所交付的内容，用户的反馈都是负面的意见。

搜索相关性在这一点上也没有什么不同之处。也许我们浪费了几个月的技术投入，认真地在实现着所谓正确的排名需求。只有在产品发布以后，才意识到用户并不像我们所期望的那样在使用搜索功能。或许购物者希望在电商网站的产品评论中



进行搜索，然而我们只实现了常规意义上的产品搜索功能。又或许有一家医院系统，其搜索返回的页面是有关疾病信息的，但实际上用户希望查找的是治疗那些疾病的医生。尽管我们付出了所有的努力，但用户对搜索的期望总是会让我们感到惊讶，总是与我们期望的行为有所出入。

本章的核心焦点是要营造一种健壮的跨职能文化，去发掘用户通过搜索想要得到的东西。我们如何才能建立这样一种企业文化，可以利用准确快速的反馈机制来矫正我们搜索应用中的那些需求呢？我们的工作如何才能持续不断地获得信息，并加以修正，从而符合用户的期望呢？

就像下棋一样，相关性搜索入门很容易——交付一个基本的、未经调优的解决方案往往是很简单的。但准确地发掘用户期望的点点滴滴，并运用技术手段对其做出快速响应，这些都把相关性搜索变成了一场激动人心的快棋（speed chess）游戏。幸运的是，与快棋赛不同，相关性搜索是一项团队运动。遍布企业的那些训练有素而身居要职的同事们都可以帮助我们了解用户需要什么，并做出适当的响应。图10.1详细介绍了一系列要素，包括：角色（相关性技术工程师、内容管理员、老板、领域专家）和解决方案（分析、测试），它们都可以为相关性工作提供反馈信息。

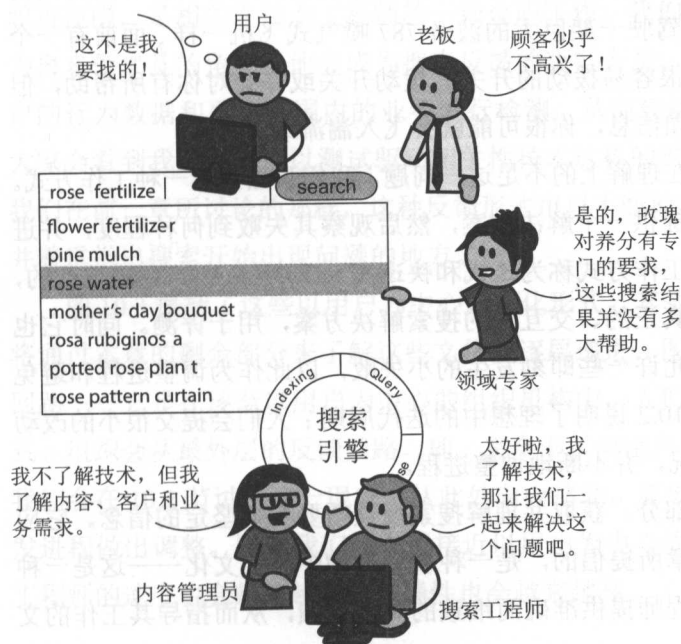


图 10.1 针对相关性技术工程师的各种反馈形式，从距离问题最远的那部分人开始（如，发现搜索结果有误的用户、老板/经理）一直到经常参与其中的那部分人（如，内容管理员和相关性测试人员）。

## 10.1 反馈：以相关性为核心的企业所依赖的基石

为了从搜索中找出用户想要的内容，我们需要承认一件事情，这也许会令人感到有些不太舒服。尽管拥有相关性技术工程师的头衔，但你可能并不知道用户是怎样看待相关性的。你可能并不知道，八岁的孩子会如何使用可检索且面向儿童的恐龙数据库的。你一定不知道，医生是如何利用医学检索应用来为病人提供诊断的。又或者，机械修理工会如何搜索汽车零件。用户使用搜索的动机是无穷无尽的。而我们理解用户及其领域的能力却是有限的。我们在黑暗中前行，没有意识到这一点也许就会给自己带来灭顶之灾。

如果不知道用户如何看待相关性，那么利用搜索的那些诱人的特性对其进行调整和激励，将会让事情变得更加糟糕。大多数开发工作都属于小修小补，比如放大处理（boosting）、更改查询类型，或者修改分析器设置等。因为这些小修小补会改变搜索排名的方方面面，所以它们会对搜索结果产生极大的影响。在不知道用户相关性需求的情况下，我们很容易就会把搜索搞得乱七八糟。我们有可能会蒙对，而且还做得很棒！但也有可能一些最简单的改动，也会对现有的那些可运行的用例构成破坏。这就像你在黑暗中驾驶一架巨大的波音 787 喷气式飞机一样，面前有一个巨大的仪表盘，上面有许多很容易拨动的开关。拨动开关或许会对你有所帮助，但如果飞行过程中没有任何反馈信息，你很可能就会飞入湍流或者更糟！

针对大家就相关性含意在理解上的不足这一问题，我们强烈推荐一种工作方式。提升相关性的最佳方式是先提供一个解决方案，然后观察其失败到何种程度，并进行相应的调整。我们将这种工作方式称为迭代和快速失败（*fail fast*）。它是迭代的，因为它专注于快速提供一个真正的、交互式的搜索解决方案，用于评测。同时它也是“快速失败”的，因为它允许一些即刻发生的小失败，以此作为调整进程和避免灾难性失败的最佳机制。图 10.2 说明了理想中的迭代周期：人们会提交很小的改动用于评估，然后观察反馈情况，并不断地调整进程。

反馈是这里面最重要的部分。获取并理解搜索反馈需要我们坚定的信念。它应该成为整个团队的信念。本章所提倡的，是一种以用户为中心的文化——这是一种尽最大努力给相关性技术工程师提供准确而细致的用户反馈，从而指导其工作的文化。

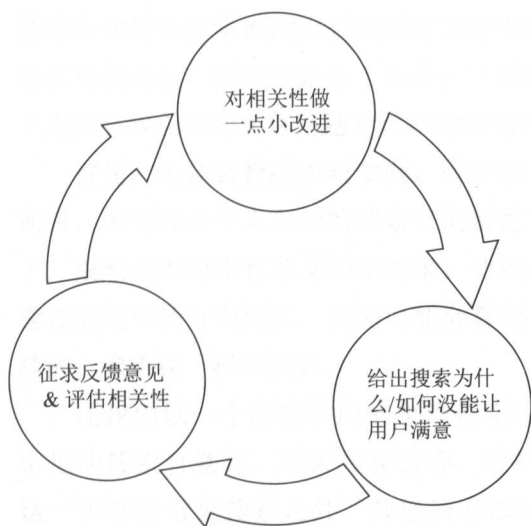


图 10.2 理想中的搜索相关性迭代：尝试简单的调整、请求反馈并为了持续改进而快速失败。

以用户为中心的文化一般会认可这样几种反馈来源。首先，它认可组织内的领域专家可以帮助我们把握相关性工作的方向。而对反馈的纠正则超出了这些专家的职责范围，是相关性技术工程师的一项全职工作。我们还会看到，有一种我们称之为内容管理员的角色，他会成为搜索反馈的决定力量。内容管理员发出指令，对用户的行为数据和更大范围内的业务进行检测，从而理解搜索结果是否正确。最终，大家会看到我们是如何以测试驱动相关性技术这样的形式来加快反馈速度的。正如我们在前一章所讨论的那样，这种反馈形式可以不断地对搜索发生的变化做出评估，并着重指出搜索开始出现问题的地方。

图 10.3 概括了这些以用户为中心的文化形式，其效果由外向内逐级加强。我们将通过本章的剩余部分来了解这些文化的逐层递进。图中的每个环形都是一个反馈回路——它是在逐渐以用户为中心的组织机构中，人们所使用的一种迭代反馈的形式。组织会从最外层的反馈回路，即：完全不了解用户的搜索需求，开始向内递进。我们会在 10.2 节讨论这一现象。从此处向内移动，逐步演进，以对相关性技术的开发进程做出调整。随着我们越来越接近以用户为中心的文化，反馈到达相关性技术工程师的速度也会越来越快，准确性也会越来越高。

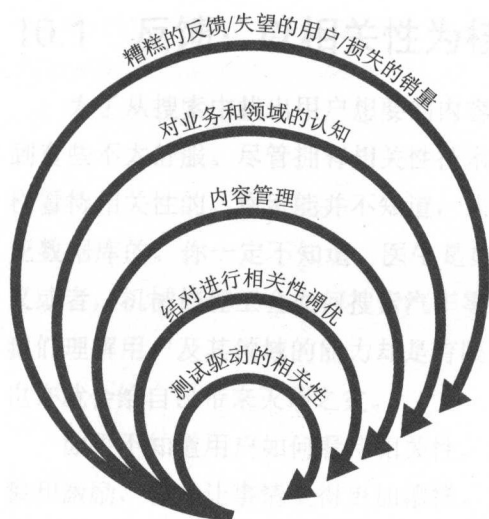


图 10.3 各种重要的反馈回路让相关性技术工程师得以对搜索的相关性做出改进。

## 10.2 为什么以用户为中心的文化比数据驱动的文化更重要

在介绍相关性反馈回路之前，让我们来讨论一下在这一点上大家可能存在的一个普遍的误解。作为一名工程师，我们可能会问，对于相关性反馈的收集问题，为什么就不能简简单单地收集那些反映用户搜索需求的海量数据，然后对其进行分析呢？

在当今世界的高端用户分析（high-end user analytics）和机器学习领域，避开文化问题是很有诱惑力的。许多人都在讨论建立一种数据驱动的文化，仅凭用户的行为数据来决定用户的使用体验。大家可能还记得，我们曾经讨论过许多这样的指标。在第 9 章中我们介绍了点击跟踪（click tracking）、深入翻页（deep paging）、抖动现象（thrashing）以及其他一些指标。有了这样的搜索指标，何苦还要咨询市场人员、领域专家，以及组织中的其他人呢？我们可以从数据中直接判别出搜索相关性的正确与否，不是吗？

我们相信，以用户为中心的文化一定优于数据驱动的文化。对于搜索而言，好的用户数据往往要么很难拿到，要么难以理解。当我们拿到了数据，通常除了相关性技术工程师以外，还需要更多人一起参与，才能分析出用户是否满意。坦率地说，

要想从数据中科学地做出有效的结论是很困难的。而要对用户的所想所感科学地做出有效的结论，则可能是难上加难。让我们来考察一下这些困难，因为它直接说明了为什么我们要把文化问题看得如此重要的原因。

首先，为什么数据很难拿到，或者说对搜索反馈用处不大呢？对于大多数应用而言，绝大部分个人发起的搜索查询就像雪花一样每片都各有不同。它们都太稀疏了，很难从中得出有意义的结论来。让我们想一想这是为什么。考虑所有用户的搜索在应用中的分布情况。我们可能会看到少量极为常见的搜索，剩下的都是大量稀疏的、像雪花一样的搜索。

让我们以一个园艺供应的电商搜索应用为例，一起来看一下这个问题。最为常见的搜索有软管、花束和花盆等，这样的搜索每分钟都会发生好几次。但除了这一小部分常见搜索之外，其他搜索都显得越来越纷繁芜杂。例如，搜索“*Rosa rubiginosa*”（一种玫瑰花的学名）可能在一年里只会出现两次。这样的稀疏搜索很可能占据了搜索流量的主体；每一条搜索都是不常见的，而这样的搜索多如牛毛！

搜索中的这种现象被称为长尾搜索（long-tailed search）。如图 10.4 所示，长尾指的是搜索项可能的分布形态。我们注意到本例中，在数量上相对较小的一组搜索项（位于左侧）占据了搜索流量的 50%，其余 50% 的搜索流量则来自于数量巨大而相对稀疏的搜索项（位于右侧）。这些搜索项构成了搜索的长尾。

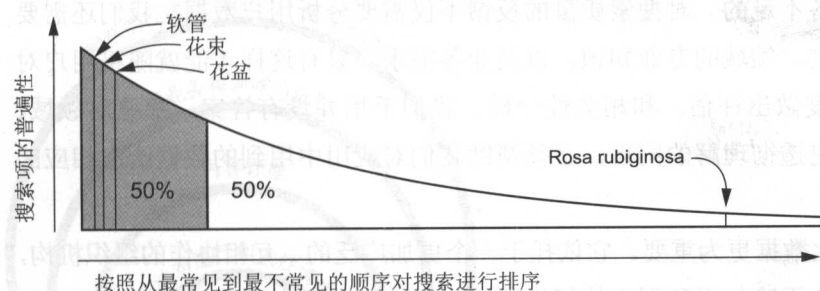


图 10.4 对长尾搜索而言，大部分流量与不太常见的搜索项有关。

搜索的长尾现象可能会导致纯数据驱动方式的相关性反馈充满挑战。如果我们的应用存在长尾，那就可能需要依赖更多间接性的定性手段才可以得到相关性反馈。在我们这个园艺的例子中，像“软管”“花束”和“花盆”这样的热门搜索，用户数据可能是有用的。但对于大多数其他类型的搜索而言（如“*Rosa rubiginosa*”“*Rosa rubiginosa Fertilizer*”），用户数据可能就会成为一种模糊不清的“注解”。长尾数据

也许能告诉我们什么样的用例对用户来说是重要的（比如用户会根据学名进行搜索），但也仅此而已。

第二个我们会遇到的问题，是如何从行为数据中获得有关用户是否满意的信息。例如，当用户不满意时，我们该如何从用户数据中获知这一信息呢？我们可以假定，用户翻到搜索结果的第二页（以及后面的页）明显是一种消极的行为。但也许用户就是想彻底查看所有得到的结果。这在专利或法律领域的搜索中经常可以看到，用户会千方百计地对每一个匹配结果都进行仔细检查。

在尝试鉴别用户是否认为结果相关时，同样的问题也是存在的。用户也许看似对搜索结果满意，因为他们购买了搜到的产品，或者阅读了搜到的文章，但“用户满意”这一结论还远不够直接。如果搜索结果的开始部分并没有出现更有价值的内容，那么用户就有可能会选择购买普通的、几乎毫不相关的搜索结果。如果用户搜的是“*Rosa rubiginosa*”，而我们为其返回的却只有雏菊（daisies），在这种情况下，用户偶尔购买雏菊这一事实并不意味着某种雏菊就是最相关的结果。不开玩笑地说，他们也许会购买一款普通的“野蔷薇（sweet briar rose）”类的产品，并且认为这就是我们这家园艺商店所提供的全部商品。与此同时，还有另一款优秀的商品埋没于数据之中，如果将它作为结果返回，那将会给商店带来更多的购买机会。

问题的关键是，即使有一大堆用户行为数据，相关性技术工程师对数据的预测工作也依然是准备不足的。对搜索质量的反馈不仅需要分析用户数据，我们还需要理解用户的上下文、领域的专业知识，以及业务需求，只有这样才能就随机用户对数据搜索的满意度做出评估。和相关性一样，我们手里并没有答案。那些对领域、业务和用户有着更透彻理解的同事，必须帮助我们对应用中用到的数据给出相应的解释。

良好的反馈比数据更为重要。它依托于一个更加广泛的、互相协作的组织机构，可以同时搜索的正确与否和用户的行为数据进行分析。只有这样，数据才能就用户对搜索的意见为我们提供有用的信息。不同于以数据驱动的文化为出发点，实际上，这才是以用户为中心的、协作型文化的、最为成熟的一种形式。通过构建这样一种以用户为中心的文化，我们将最终上升到一个新的阶段，让数据可以发挥其应有的作用。



## 10.3 无视相关性的天马行空

通常，我们一开始会遇到一个较为普遍的问题：组织机构对搜索的相关性一无所知。组织成员并没有把相关性视为他们成功的基础。与快速失败的迭代形式不同，我们经历的唯一一次迭代是这样一段痛苦的历程：大家开发出完整的搜索解决方案，将其交付给用户使用，然后痛苦地获悉用户对交付的结果十分失望。

遗憾的是，我们有许多同事可能都以为搜索引擎有某种内置的类似谷歌那样的智能逻辑。他们可能会认为搜索引擎知道什么是最佳的结果，返回的内容不会有错，也无须配置或编程。对许多人来说，用户对我们的应用有独特的相关性预期，这样的说法真的是闻所未闻。

在我们称之为无视相关性（relevance-blind）的企业里，缺乏对相关性的认知，有可能会让我们的组织处于危险的境地。身处这样的环境，会把我们暴露在反馈回路的最外层：销量持续下滑，客户怨声载道，如图 10.5 所示。在这个阶段，由于搜索出现的问题，导致用户停止了他们想做的事情（例如，停止了商品的购买行为）。在发布或更新了产品之后不久，组织才突然意识到搜索相关性的重要价值，这种情形并不少见。遗憾的是，只有在用户的参与和产品的销量突然骤减之后，人们才会有这样的意识。



图 10.5 组织缺乏对相关性问题的认识。团队如果只在最外层的反馈环上工作，就有可能发现搜索表现不佳：销量下滑，客户抱怨。

怎么知道自己的组织是否处于这样的状态呢？一般来说，这种无视相关性的组织有几个共同的特点。我们会注意到，其中最主要的特征是搜索工作不涉及任何相

关性的工作。相反，人们把重点放在了功能、性能，以及规模化扩展（scaling）上了。我们可能会被当作没有专业的相关性调优经验的后端搜索工程师来看待，而不是专门的相关性技术工程师。当搜索成为用户体验的核心组成时，这样的做法是有问题的。人们看到的是又一帮脱离用户及其需求的后端开发人员，这对于用户体验的打造没有任何帮助。另外，像这样的组织，其危险之处在于，它可能会有意识地让我们对用户的真实目标避而不谈。相反，企业所希望看到的，是我们能让搜索再快 10 毫秒，或者能找到办法集成新的文档。

大家很容易猜到发生这种情况的原因是什么。搜索引擎看上去就像是又一种数据存储。操作搜索引擎与操作像数据库这样的系统有许多共同的特征：两者都必须持续不断地对数据进行存取。它们必须保持可用性和高性能，为我们的应用存储所需的数据。一开始，许多人可能并没有对搜索引擎中稀奇古怪、看似神秘的排名功能产生疑问或认真去想。但搜索引擎与我们所了解的其他应用是有所不同的，而许多人则更愿意把它当作他们所熟悉的数据库来看待。

正是由于这种无知，当我们在与应用打交道的时候，相关性排名就成了一个盲点。如果给一个典型的搜索应用截一张图，我们会发现，搜索结果就在页面的正中央。但在一个无视相关性的组织中，如图 10.6 所示，前端开发人员和设计人员只对搜索应用的功能组件精雕细琢，而不去关心其相关性排名的正确与否。许多功能的开发和错误的修复都是围绕绝大多数典型应用开发中的那些常规套路展开的：登录 / 登出、外观效果（look-and-feel），以及其他自定义功能。当说到搜索，团队则更愿意去改进应用中那些比较容易理解的功能组件：切面（faceting）、过滤（filtering），以及其他浏览功能。或者，他们也许会去改进搜索结果自身的展现形式——而不会去理会搜索的排名效果如何。

这样的组织经常会在应用发布以后幡然醒悟。用户第一次键入他们的搜索查询，却找不到想要的内容。在我们的咨询工作中，不止一次目睹了企业无视相关性带来的后果。这样的后果从来都不会好到哪儿去：客户十分失望，用户涌向了竞争对手。尽管应用开发人员和设计人员都很努力，但直接来自用户的绝大多数反馈信息却是：“搜索做得烂透了”。

此时此刻，企业进退维谷，于是团队找到了你这位搜索工程师，想知道该如何渡过难关。突然间，你不再是一名简简单单的后端开发人员了。在处理用户体验的问题时，你必须发挥核心的作用！我们此前曾经讨论过，如何帮助相关性技术工程

师为其工作获得更多及时的反馈，这一问题成了应用成功的关键。现在就开始寻找“快速失败”的方法。在组织遭受下一个尴尬的错误之前，我们需要找到办法，精确地获得相关性反馈。

在无视相关性的企业中进行搜索

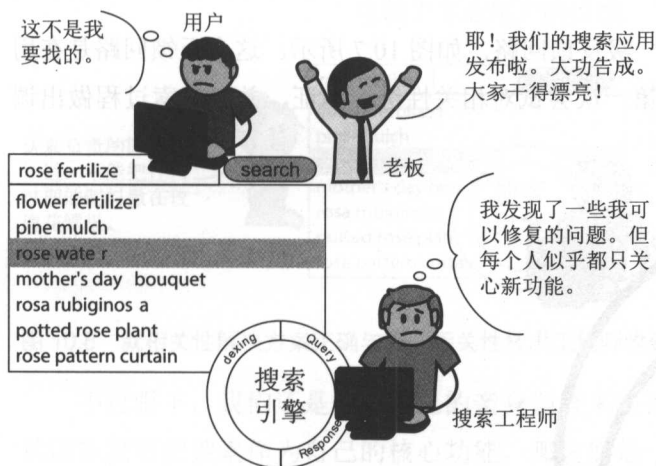


图 10.6 在一个无视相关性的组织中，似乎没有人注意到搜索返回的结果不尽如人意。相反，团队却在忙于其他更为轻松的任务。

## 10.4 相关性反馈的觉醒：领域专家和专业用户

一下子接到任务，就要求我们判断出用户在搜索排名之外还想要些什么，这也许会有点难度。作为一名相关性技术工程师，我们不会对每一类用户的期望都了如指掌。初涉相关性工作，也许你会觉得，相比于对用户体验的关注，自己在数据库和代码领域会更加得心应手。更糟糕的是，用户可能来自某个特殊的领域，比如医学或法律。像“Myocardial Infarction（心肌梗死）”这样的搜索对你而言可能毫无意义可言。最后，企业可能有其自身的排名预期，他们希望通过搜索对用户加以引导——将用户导向利润更高的产品，或信息量更大的网页。我们需要在更大范围内从组织中寻求反馈，以对搜索的步骤做出调整。

为了探寻相关性对应用的含义，我们必须成为这一领域的权威。为了建立相关性反馈的第一个回路，我们需要得到组织的认可。很快我们就会意识到，我们所要的信息散布在整个企业的各个角落。根据组织类型的不同，对于“用户想要什么”

这一问题的定义在市场、销售、QA（质量保证）、法律或医学分析、管理，以及许多不同类型的群体之中可能会大不相同。我们必须成为跨职能协作的行家里手。我们必须成为高度社会化分工的推动力量，而不是一时心血来潮地写上几段好玩的代码就完事了。要从搜索体验中找出用户期望的是什麼，就必须打破组织的壁垒，找到我们一直在寻求的答案。

现在我们要做的是构建下一级反馈回路，如图 10.7 所示。这个反馈回路是在用户看到失望的结果之前，组织第一次尝试对相关性进行验证，并对搜索过程做出调整。

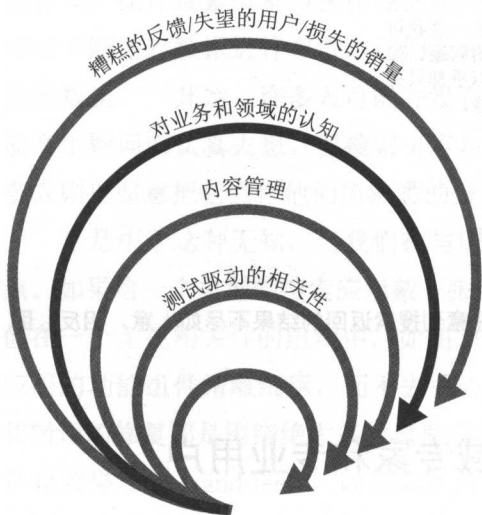


图 10.7 企业内部的第一个反馈回路：临时从 QA、BA 和其他专业的独立团队获得的相关性反馈。

除了那些对用户有所研究，以及和用户密切合作的团队之外，我们应该首先考虑邀请一位专业用户。专业用户曾经是用户群体中的一员，但现在为我们的组织工作。如果我们正在搭建一个园艺网站，为专业的园艺师提供便利，那么市场、QA，或者组织内其他团队中的相关人员也许可以充当园艺专家。专业用户是搜索反馈的“聚宝盆”。他们可以直接模拟用户在遇到不相关的搜索时所相应给予的负面反馈。只不过与大多数用户有所不同的是，这些用户和我们的成功密不可分。他们更有可能为我们提供详细而有建设性的反馈信息。图 10.8 显示了一位专业用户正在指导相关性技术工程师如何得到更好的搜索结果。

企业或许也会做可用性测试（usability testing）。利用可用性测试，人们可以在目标用户群与应用进行交互时对其进行研究。对用户行为的跟踪可以达到很深的程度——包括视线跟踪和表情监控等。用户也可以提供有关应用的一些定性的反馈，

就如何看待应用等一系列问题给出答案。或许我们可以借助这样的测试，得到应用在搜索相关性方面的反馈。我们甚至可以与这些用户进行交流互动，确保他们所获得的结果对系统的相关性排名是有所帮助的，更重要的是，对于他们没能获得所需内容的情况——我们也可以从中获得与专业用户同样丰富的反馈信息。

依赖于专业用户的反馈

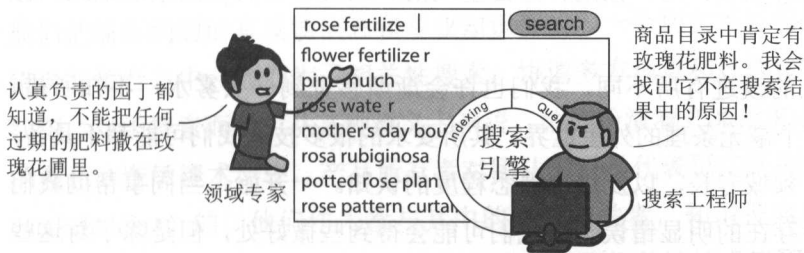


图 10.8 就相关性解决方案正确与否，相关性技术工程师收到了来自领域专家的反馈。

不过眼下，我们还是通过自己的亲身调查来获得相关性反馈的。如果这些封闭的团队没有把搜索作为自己的核心功能，那它们是一点用处都没有的。而其他团队则有他们自己的工作要做，因此只有有限的精力能够为我们提供帮助。所以，我们的测试仍然是临时性的、非正式的，而且还要取决于其他团队的成员是否有时间。

因而，即使我们坚持采用这样的策略，它依然是一种信息量少而反馈缓慢的回路。其所提供的反馈，有些时候是颇有深度和极具价值的，而有些时候却作用有限。虽然许多问题得到了解决和避免，但还有一部分问题依然存在。尽管如此，这一阶段的输出仍然是至关重要的。通过打破技术上的壁垒，我们也许能和其他职能团队之间建立起强大的联盟。大家开始看到搜索对用户的重要性了。即使受到的关注有限，你和你的盟友可能已经开始认识到，设立一个全职的角色来对搜索质量进行管理是很有必要的。

## 10.5 相关性反馈的成长：内容管理

在 10.4 节中，我们讨论了相关性技术工程师作为处理相关性反馈事宜的专家，为了明确组织对相关性的定义，他们：打破壁垒、四处奔走、挨家挨户地去敲门，利用这种原始的方法，尝试建立初始的反馈回路来指导我们的工作。遗憾的是，我们不能将反馈收集一直维持在这样一种水平，然后完成我们的技术工作。而且，同

事们也并不总是有时间能给我们提供持续不断的、高品质的搜索反馈。

更重要的是，我们从同事那里得到的答案会逐渐变得越来越不一致。关于搜索应该做些什么，要找到“正确的答案”，我们越来越需要协调多方观点才行。市场部可能会要求我们基于公司和广告商与供应商之间的关系搞些促销，而领域专家则可能会建议将专业高端用户的需求放在比普通用户更高的位置。要得到“正确的答案”，可能要取决于我们对领域、用户预期、办公室政治，以及许多其他因素的细致入微的理解。

和驱动技术上的改进有所不同，我们也许会渐渐感觉到一头雾水，不知所措。我们仿佛置身在一个毫无条理的外星世界，其所要求的很多技能我们可能都不具备：比如对业务需求、领域专长，以及用户熟悉程度的认知。一开始，当同事帮助我们指出了搜索结果中存在的明显错误时，我们可能会得到些微好处，但是除了对这些明显的错误进行修复以外，要弄明白搜索应该怎样工作才算合理，这项工作就落到了我们自己的肩上——它使我们的注意力从自己所擅长的关键性技术工作转到了其他地方。

企业解决这一问题的方法是增设一个专门负责搜索反馈的角色：内容管理员，如图 10.9 所示。

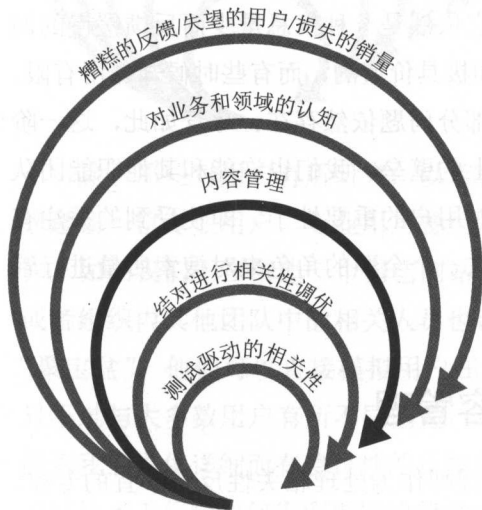


图 10.9 内容管理，就是为相关性技术工程师，针对来自企业中、更大范围内的反馈信息进行管理的一种行为。

### 10.5.1 内容管理员的角色

内容管理员负责接受所有形式的反馈，并确定“正确的答案”是什么。一位理



想的内容管理员拥有足够丰富的业务/领域专业知识、熟悉用户的情况，并且有资格判断搜索的正确与否。内容管理员会认真地思考正确的搜索行为。他们对于用户如何与搜索应用进行交互有着深刻的理解，并且能将其翻译成可执行的反馈，以供相关性技术工程师加以实现。

内容管理员这个角色要与更多的业务打交道。他们也负责对任何可能出现的用户行为数据做出解释。他们明白业务（不仅仅是用户）想要从搜索中得到些什么。他们把捕捉到的所有反馈信息和专业知识都放入一个切实可行的功能改进与待修复错误的集合之中。他们为了相关性搜索，协调多方观点和行政策略，形成统一的路线图。对于搜索而言，从某种意义上来说，他们扮演着产品所有者（product owner）的角色。在敏捷术语中，产品所有者对团队来说是代表更广泛的业务和利益相关者（stakeholders）的。他们作为参与其中的利益相关者，知道搜索应该如何工作，并且对于搜索的研发方向，他们为技术团队——即相关性技术工程师们——指明了更为宽阔的方向，如图 10.10 所示。

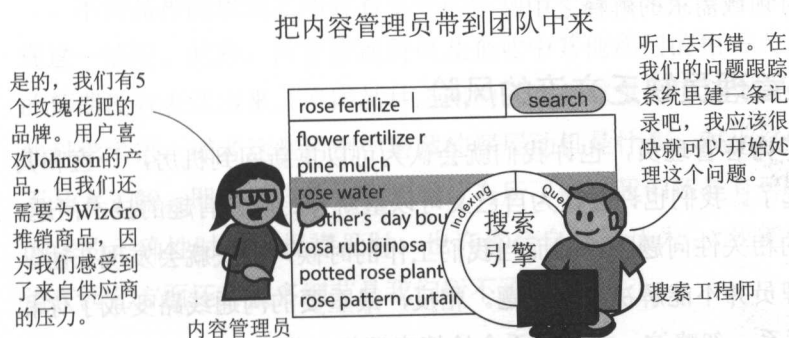


图 10.10 内容管理员基于对广泛业务的反馈/咨询，以及对用户反馈的评估，为搜索行为给出定义。

一些内容管理员也许多少有些技术背景，可以在一定程度上实现相关性的自助服务。他们也可以帮忙对信息进行组织和管理，这些技能可以为搜索带来许多价值。我们在第4章中讨论过如何在搜索中使用同义词（synonyms）和分类法（taxonomies）。通过把不同的搜索词彼此关联，我们可以利用这些工具来显著地提高搜索结果的相关性。由于内容管理员拥有相关领域的专业知识，因此他们可以帮助我们对这些概念层面的关联关系进行管理。

一个园艺类的搜索引擎，能够把“Reel Mower”理解为一种“割草机（Lawnmower）”，或者知道“Lawnmower”与“Grass Cutter”是同义词，这样的搜

索引引擎，可以极大地帮助提高搜索结果的相关性。我们自己不太可能具备足够的技能，可以在草坪护理类产品的分类体系中将这些概念进行关联。但是一位内容管理员也许能够按照用户所希望的，搜索应用能够理解的行业术语，来对这些关联关系进行分类。

因此，一位理想的内容管理员对相关性解决方案是有所帮助的。为此，我们只要让内容管理员根据各种排名因素对放大值（boost）和权重值加以控制就行了。回忆一下第9章中 Yowl 的例子，我们曾经在内容、餐厅位置、价位偏好，以及其他因子之间寻求平衡。有了技术基础作为依托，还有谁能够比内容管理员更适合于寻找这些因子之间的最佳平衡呢？

为团队加派一位内容管理员对我们的相关性工作可以起到事半功倍的效果。内容管理员可以令团队扬长避短。他们在组织机构里四处游走，仔细聆听用户的反馈。他们可以对搜索的过程提供可靠而明智的修改意见，帮助勾勒出“画作”的轮廓，以便于我们用技术细节对其进行着色。他们可以让我们专注于自己的技术技能，而不是深陷在政治和对领域需求的解释之中。

### 10.5.2 与内容管理员缺乏交流的风险

一旦团队里有了内容管理员，也许我们就会认为可以重新回到机房，一直和其他程序员们待在一起了。我们也许会认为自己又可以重新开始研究有趣的技术问题，修复进入问题队列的相关性问题了。然而当我们工作的时候，很快就会发现团队中拥有一位内容管理员并不能解决所有问题。相反，最重要的沟通线路变成了你和内容管理员之间的联系。忽略这一关联关系会给搜索调优工作带来混乱和麻烦。

当内容管理员给我们分配新的任务时，我们可能会建立起一种不正确的工作流程：我们很少开口说话。相反，在团队中盛行的，是通过电子邮件和问题跟踪系统这样一种缓慢且有信息损耗的反馈渠道进行沟通。我们可能会从工作队列中取走排在前面的若干问题，将它们逐一处理完毕，也许我们会认为这些问题相对独立且易于理解。然而不幸的是，解决相关性问题的很少会如此简单。我们需要对解决方案进行频繁的后续沟通，并让内容管理员进行评估。

例如，假设我们还在为园艺师开发园艺类的搜索应用，或许作为相关性技术工程师的我们拿起一张纸条，上面这样写着：

通过植物的学名（*Rosa rubiginosa*）对其进行搜索，应该与按非学名（sweet briar rose）的方式进行搜索返回相同的搜索结果。

哈，够简单吧！内容管理员提供关键词的等价信息，而我们则添加几个简单的同义词。当我们宣布任务完成以后，又过了几天，我们收到了来自内容管理员的另一封邮件：

Hi, 相关性工程师，

我发现搜索“*Rosa Rubiginosa*”会返回其他品种的玫瑰，比如“*Rosa Glauca*”，而不只是某一种特定的玫瑰。另外，我希望在搜索普通的“*rosas*”时能够显示包含不同品种的搜索结果，而不是让结果看上去好像特别偏向于某个品种。搜索“*Rosa*”得到的结果是不是应该与“*Rosas*”一样呢？这样做可行吗？让我们来讨论一下吧……

不同品种的玫瑰之间有着重要的区别——园艺爱好者和内容管理员很快就会发现这一情况。此外，内容管理员以及企业中其他部门的相关人员，都还有一些隐含的预期没有表达出来。不同品种之间的区别在哪里，用户会如何使用“*Rose*”的这些科学术语，或者这些用例中潜藏的深层动机是什么，对于这些问题，我们可能并不是太了解。即使用户认为我们对此有所了解，我们还是有可能在实时评估搜索方案的正确性时感到捉襟见肘。也许你会自信地认为：“我能做到！”然而，要知道在园艺方面还有很多细节是我们所不了解的！

## 10.6 让相关性更加流畅：工程师/内容管理员的结对

为了能够对搜索的相关性进行适当的调优，我们需要经常与内容管理员进行面对面的交流，从而了解用户对搜索应用的行为预期。这种交流的途径要比偶尔聊天来得更加深入。我们鼓励大家在解决相关性问题时能与内容管理员并肩工作。

大家也许听说过结对编程，就是指两名程序员在一起工作，共同解决一个编程问题。在这里，为了提高搜索的相关性，内容管理员和相关性技术工程师也要参与结对，一起对搜索进行调优，并且是在同一张办公桌上一起工作。这就是我们下一个层级的反馈回路，如图 10.11 所示。



图 10.11 下一个反馈回路：内容管理员和相关性技术工程师一起工作。

通过结对调优，内容管理员和我们都坐到了“相关性搜索”这架大型喷气式客机的驾驶室里！我们对搜索的微调可以立即由内容管理员进行评估。一些简单的错误可以即刻被发现，并通过对搜索过程的简单调整加以修正。与之相反，在尝试了既费脑又费时的邮件交流以后才发现问题的所在，这样的做法会白白浪费我们宝贵的时间，并使我们远远偏离了正确的进程。双方都要把对搜索的调优看成是一个高度协作的迭代环，它要求我们认真运用两种不同形式的专业知识。图 10.12 显示了内容管理员和相关性技术工程师应密切合作，共同解决相关问题。

为相关性进行结对

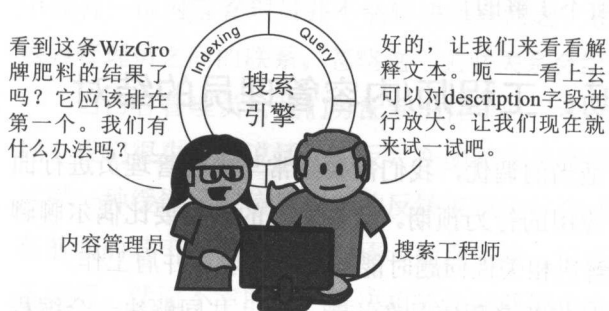


图 10.12 当相关性技术工程师和内容管理员并肩工作时，他们可以很快地解决问题。

最重要的一点是，在结对时，两个人可以共同探索各种可能性，而不仅仅是按照要求去完成任务。与简单的任务分配不同，内容管理员在表述问题的时候也许会有深度。而我们则或许可以认真地思考一下，在“搜索空间（search space）”中

由各种解决方案构成的、广泛意义上的生态系统，以解决各种类型的搜索问题。此处的沟通也可以换一种方式展开：我们可以逐步让内容管理员知道，哪些搜索任务简单，哪些则较为复杂，通过某些功能的改进从中快速地识别出何时需要折中，怎样做容易成功。

从全局来看，我们已经从追着别人要反馈，逐渐转变成现在所倡导的，不但设定全职的角色，而且还要采取近乎全职的结对方式。我们的反馈回路被压缩成一种更加紧凑和更为直接的反馈形式。这种反馈形式反映出了相关性工作的迭代特征。相关性技术工程师常常可以对搜索引擎的行为做出快速修改，而“尽快尽早地失败（failing fast and early）”这一点是很重要的。通过提供更加紧凑和更为直接的反馈形式，我们的工作也越来越接近正确的结果了。现在，当我们驾驶着这架巨型喷气式飞机时，至少可以看清迷雾，并使自己保持在安全的飞行高度了。

## 10.7 让相关性加速：测试驱动的相关性

与内容管理员的结对可以解决重要的沟通问题。然而我们很快就会意识到，修改相关性时，即使做了最好的打算，也往往会产生新的问题。对搜索的调优，起初感觉就像是在玩“打地鼠（Whac-a-Mole）”的游戏。我们专注于一系列问题的解决。但解决这些问题，会导致现有的、工作正常的搜索出现质量下降的情况。在我们有关园艺的那个例子中，比如我们也许会彻底解决根据学名（*Rosa rubiginosa*）进行搜索的问题。但不幸的是，如果我们对园艺一窍不通，就有可能为热门搜索返回毫无意义的结果。对“Rose”的搜索就有可能优先返回“雏菊（daisies）”的结果！（而它的确是一种玫瑰花的名字！）

### 10.7.1 理解测试驱动的相关性

当对搜索进行调优的时候，我们需要让自己和坐在副驾驶位置上的内容管理员都能对搜索的过程进行有效而实时的修正。为了让反馈变得更加广泛（broad）、及时（instant）和定量化（quantitative），我们能做些什么呢？

答案来自传统软件开发中的测试驱动开发领域。在这些方法论中，我们要构建出一套可以运行我们的代码的自动化单元测试。这些测试根据我们对应用正确性的理解对代码的输出做出评估。失败的测试会告诉我们哪些功能有问题。有了足够的测试覆盖，我们就可以去实现新的功能，修复错误，同时还能确保旧的功能可以继

续工作。

测试驱动的相关性，如图 10.13 所示，在我们对搜索进行调优时，利用自动化测试对搜索解决方案做出评估，从而解决了“打地鼠”的问题。我们依然需要内容管理员和其他同事帮助给出搜索方案正确性的定义。理想情况下，这些同事会帮助我们建立自动化测试。这些测试表现为一系列重要的搜索查询，能够以程序的方式加以运行，并对运行结果的质量与正确性做出评估。这种做法有效地把专家的智慧“封箱打包”，让我们能够基于专家的反馈，自动化地对搜索结果做出评估。

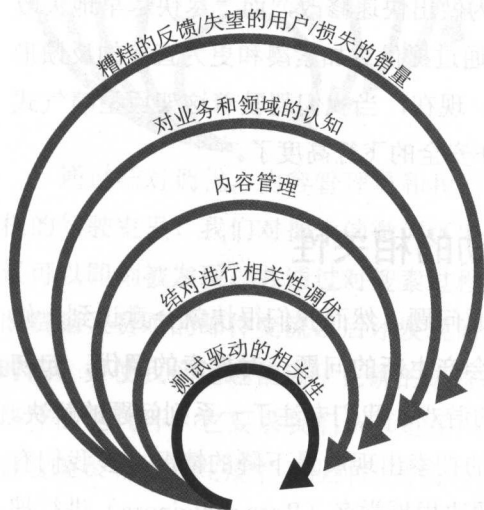


图 10.13 相关性反馈最直接的形式：测试驱动的相关性。

有了测试驱动的相关性，结对调优就有了新的工作内容。与通过手动测试来改进相关性有所不同，你和内容管理员都要着手负责对相关性测试的维护。如果一个针对“*Rosa rubiginosa*”的修复破坏了对“*Rose*”的查询，我们立刻就会发现这一问题，并且能够根据与用户和业务相关的用例，做出明智的决策。

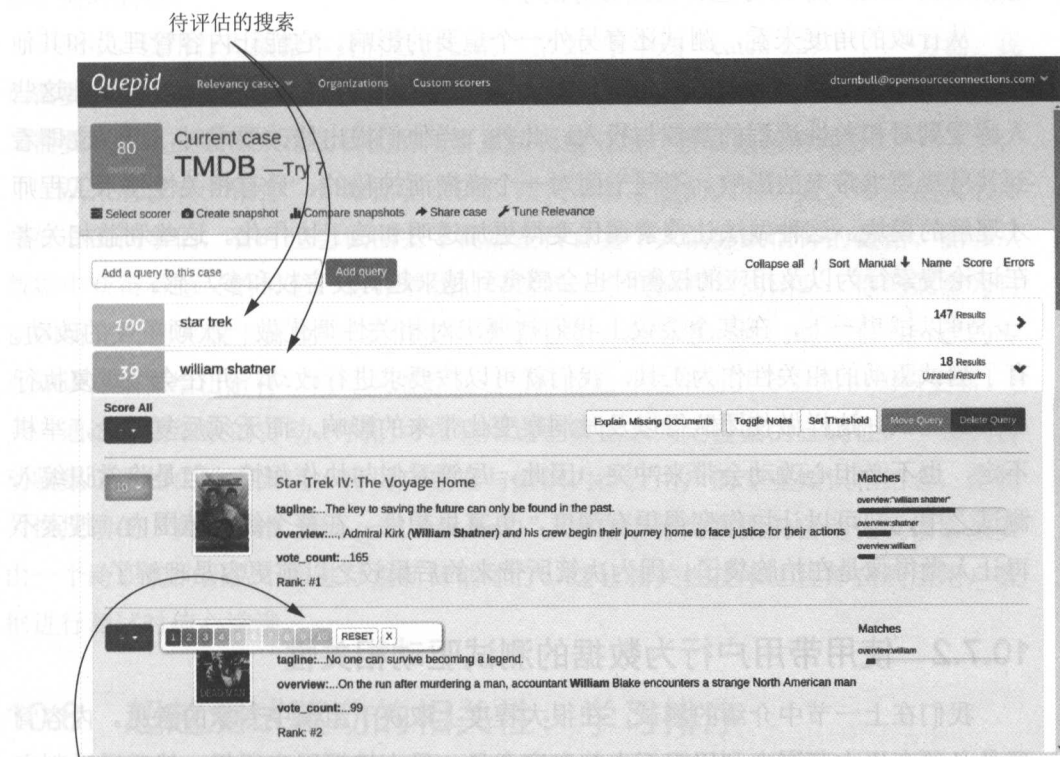
相关性测试一般都采用什么样的形式呢？对于相关性搜索来说，测试大体有以下两种形式。

- 判定列表 (*judgment lists*) ——我们为每一条搜索查询的返回结果都定一个等级或给一个判定。例如，搜索“*Rosa rubiginosa*”的用户也许会把 *sweet briar rose* 的级别定为“完全匹配 (*exact match*)”，其他涉及 *rose* 的匹配结果有可能会被定为“OK”级别，而所有其他的结果则被定义为“*poor*”级别。



- 基于断言的测试 (*assertion-based testing*) —— 不同于去收集针对结果的明确判定，此处我们会在传统的单元测试中增加更多专门的断言。例如，可以设置这样的断言：当用户搜索“*Rosa rubiginosa*”时，返回的第一条结果应该是一种叫“*sweet briar rose*”的植物。

图 10.14 显示了如何利用 Quepid (透露一下：开发 Quepid 应用的正是本书作者之一) 来收集相关性判定。内容管理员利用 Quepid，根据查询的相关性表现，对搜索结果进行分级。这样做，可以对当前搜索解决方案的查询产生一个整体上的质量评价。



内容专家对该结果的相关性给出判定

图 10.14 Quepid (<http://quepid.com>), 一个针对 Solr 和 Elasticsearch 的测试驱动的相关性产品, 可以看到我们在这里使用了判定列表。

虽然想法不错，但要维护这些判定结果却需要极大的成本。随着数据的变化，我们必须不断给出新的判定。这项工作变得繁重起来。此外，它还不太容易被外包

出去。因为这些判定需要我们对领域知识具备一定的基础才行。许多组织聘用大量的相关性测试人员，其唯一目的就是要维持相关性判断的准确性。

与判定列表相反，基于断言的测试允许有更多针对性的黑白测试（black-and-white testing）。传统的单元测试不像我们保持判定的准确性那样需要昂贵的维护成本。相反，我们只需简单地指明判断“正确性”的标准就可以了。有许多组织靠的就是这种更加简单、更容易维护的单元测试。不过简单的测试无法帮助我们轻易看出不同查询之间在相关性质量上的详细分级情况。我们没有办法让一个测试只通过75%。而利用评估得到的判定结果，却可以让我们了解到，一条查询大概能达到理想预期的75%。而75%也许已经够好的了！

从行政的角度来看，测试还有另外一个重要的影响。它能让内容管理员和其他利益相关者感觉到自己也在深入参与相关性工作。通过帮忙定义测试，可以让这些人感受到对相关性的掌控与投入。此外，当他们提出修改要求时，可以立即看到其所提要求带来的影响。不同于面对一个模糊而神秘的、只有相关性技术工程师才理解的模块，这种做法让搜索调优变得更加透明和趋于协作化。这些利益相关者在讨论搜索行为以及相应的权衡时也会感觉到越来越有发言权和参与感。

可以试想一下，在某个会议上我们被要求对相关性的调优做一次颠覆性的改动。有了测试驱动的相关性作为工具，我们就可以按要求进行改动，并在会上重复执行必要的测试。这样做使团队得以及时洞察变化带来的影响，而无须反复讨论，举棋不定，也不必担心改动会带来冲突。因此，尽管看似与协作相悖，但是将常识编入测试之中，却可以让协作变得更有深度，更富思想性。在整个组织范围内，搜索不再让人觉得像是在拍脑袋了，因为决策所带来的后果较之以前更容易理解了。

### 10.7.2 使用带用户行为数据的测试驱动相关性

我们在上一节中介绍的测试，在很大程度上取决于组织中专家的意见。内容管理员必须在更大范围内利用组织内的专家意见，尽力挖掘用户所想，然后通过判定列表或测试将其编入测试工具。内容管理员完全依赖于这些同事们所提供的信息能够正确无误，并且他们在判断上不存在误解、无知，或一己私念。

因为同事的观点未必准确，所以参考用户行为数据（例如，用户点击了什么或购买了什么）得出判定列表的做法变得越来越有吸引力。在第9章中，我们讨论过搜索中一些特定的用户行为模式，如，抖动现象，这些模式可能说明了用户对结果

的失望。内容管理员也许可以利用这些模式识别出失望的用户。他们也许可以进一步识别出用户什么时候会扎堆，或者做一些对自身或业务有价值的事情，比如购买商品或订阅简报（newsletter）等。然后，我们就可以将这些信息手工或自动地翻译成相关性判定了。

这一途径看起来似乎是不言自明的，但即使有大量关于用户如何使用搜索应用的数据，挑战依然是存在的。要想理解如何把行为数据转化为相关性搜索的指标，以及这些指标中缺了哪些，仍然需要我们具备有关用户的专业知识。那些帮助判定相关性的同事们，往往也需要它们来参与决定，哪些指标适合用来判断用户何时失望，何时满意。

所以，使用行为数据本身有它自己的“第22条军规（catch-22）”<sup>1</sup>。一方面，我们需要一群不完美的人类同事通过行为数据帮助了解用户的想法。与此同时，我们又需要利用这些数据来帮助我们的同事重新调整对用户搜索预期的理解。

从来就没有“银弹”！数据驱动的反馈夹带着大量的人为因素。即使是我们那些好心的同事们，有时也会试图去寻找那些符合其个人偏好或信仰的数据。能够从数据中获得有意义的信息，意味着我们有这样一个充满智慧的团队，他们愿意为自己的所见所闻“放下身段”。我们要的就是这样一众同事，他们既能利用自己的专业知识对数据给出解释，同时又愿意通过用户的意外之举对其专业知识加以更新。

通过辛勤细致的劳动，我们可以利用这些数据来判定搜索的正确性，一直到自动收集这些相关性的判定。将用户数据与组织内涉及用户的专业知识相结合，是相关性反馈的“黄金标准（the gold standard）”。来自于行为数据的自动化反馈必须要由一个跨职能的团队不断进行重新评估才行。这让相关性技术工程师在利用用户数据进行测试时信心满满。

## 10.8 超越测试驱动的相关性：学习排序

有了团队通过努力从数据中得来的信息，另一些旨在提高相关性的自动化实践就变得颇具吸引力了。一些先进的组织机构可能已经开始利用机器学习技术来预测搜索结果与查询条件的相关性了。这种自动进行相关性处理的实践被称为学习排序（learning to rank）。

<sup>1</sup> 此处指一种左右为难的境地。——译者注

有了学习排序，相关性判定，无论是人工的还是自动的，都可以被用来学习：我们的内容当中，有哪些特征可以从全局角度对相关性做出预测。这一前沿领域正在逐步走出信息科学研究的象牙塔，出现在一些前卫的搜索应用商店里。<sup>1</sup>

在如图 10.15 所示的反馈回路图中，学习排序成为测试驱动相关性内部的一个核心热点（the red-hot center）。它是建立在所有外围反馈回路的基础之上的：它依赖于一个以用户为中心的组织，知道如何解释用户数据，以对相关性做出评估。由于学习排序是现在的一个热门话题，因此这一点尤为重要。在很多人看来，我们似乎可以不必大费周章地去建立一个密切了解用户的组织了。但遗憾的是，没有任何捷径可以避免本章中我们所讨论的这些工作。我们仍然需要一个能够深入关注用户搜索需求的组织：一个能够在用户行为数据中找到相关性标识（markers）的团队。从数据中获取与搜索用户体验真正相关的信息，仍然是一个文化和技术上的难题。

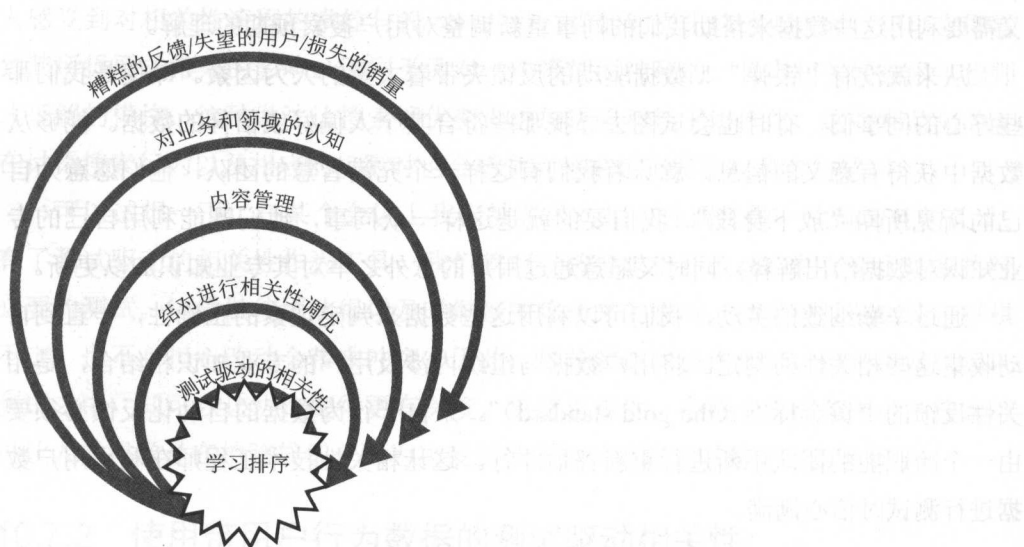


图 10.15 学习排序是一种相关性调优的前沿方法，有望在将来的某一天能自动得出理想的相关性参数。

最后，我们需要把本章学到的所有知识与前沿的信息检索和机器学习技术结合起来。通常，利用本书前面讨论过的那些简单的技术，我们可以得到较为简单的相关性收益。在我们的咨询工作当中，通常一个很简单的调整就可以为组织带来更为

<sup>1</sup> Solr 的读者也许会对下面这个补丁感兴趣，它提议把学习排序的功能加入 Solr: <https://issues.apache.org/jira/browse/SOLR-8542>。

直接且风险更低的收益，在此之后，我们才会受雇去实现更为高阶的解决方案。因此即使没有数据分析师，我们也可以针对分析器（analyzer）或查询策略给出简单的调整，借助测试驱动的相关性，在搜索的最终效益上取得非常显著的提升。

不过，有了正确的专业知识和数据以后，学习排序在帮助我们突破相关性调优的“边际效益递减规律（diminishing returns）”方面，可以发挥出巨大的威力。它以代表相关程度的用户交互数据为基础，对搜索实施快速迭代。一旦万事俱备，我们就可以着手考虑这一项令人振奋的前沿技术了。

## 10.9 本章小结

- 搜索的相关性工作的高度迭代的，且当我们通过快速失败（failing fast）来寻求反馈时，其表现最佳。
- 用户的行为数据（如点击和购买）并不是万能的，以用户为中心的文化应该优于数据驱动的文化。
- 无视相关性的组织缺乏对搜索相关性的认知，并且经常会在相关性的糟糕表现导致订单流失和用户抱怨的时候，感到手足无措。
- 相关性技术工程师从同事那里寻求反馈，开始着手解决相关问题。这些同事大致都是对用户有所了解的，包括：领域专家、QA 和销售人员。
- 内容管理员可以通过为相关性技术工程师提供反馈信息，帮助对搜索进行优化。理想情况下，相关性技术工程师和内容管理员在相关性问题上应该是并肩工作的。
- 测试驱动技术可以帮助实现相关性判定的自动化，但这些技术仍然要有赖于内容管理员，以及其他关注用户的专业人士。
- 要把反映搜索使用情况的数据结合进来，就需要我们对其做出详细的解释，而企业对用户使用搜索情况的理解，也可以通过这些数据加以补充和修正。
- 学习排序技术可以实现反馈过程中一部分工作的自动化，但是它依然有赖于我们对行为数据的维护和解释。



# 语义和个性化搜索

## 本章要点

- 针对每一位用户让搜索更具个性化
- 根据含义而不仅仅是单词来对文档进行匹配
- 将推荐作为一种广义的搜索加以实现

我们即将要结束这段漫长的旅程了。通过前几章的学习，我们已经学会了如何利用搜索技术来构建具有相关性的搜索应用，但这还是不够深入。在本书的最后一章，我们将把视线投向更远的地方，探讨一系列更加前沿的实验性方法，以此来提高用户的搜索体验。特别是，在这里我们将要讨论两种彼此相关的技术，这两种技术能够给我们带来更好的相关性。

- 个性化搜索 (*personalized search*)，利用对用户的认知，提供满足该用户特定口味的定制化的搜索结果。用户的信息可以通过用户与系统以往的交互，以及用户直接告知系统的任何信息收集得到。
- 概念性搜索 (*concept search*)，根据从文本中提取得到的概念，而不是仅仅依靠单词，来对文档进行排名。概念性搜索依赖于对搜索领域的深入了解，包括该领域中出现的术语以及概念之间的关联关系。



当我们结合使用这两种技术时，搜索方案既能理解用户的个性化需求，同时也能够领会内容中潜在的含义。

构建一个良好的个性化搜索或者概念性搜索需要做大量的工作。我们应该将本章中提到的方法看作从概念角度出发的一个起点。但是请注意，这些方法面临着一些技术上的风险；它们可能实现起来很困难，而且通常要一直到这些方法实现完毕之后，我们才能知道其效果到底如何。尽管如此，对于现有的搜索应用而言，这些方法是值得我们去仔细研究的，因为它们可能会带来巨大的价值。在接下来的讨论中，我们将给出几种个性化和概念性搜索的实现思路。在有关这两类搜索的讨论中，我们会从一些相对简单的方法开始，然后概括介绍基于机器学习的、更为复杂的方法。

在介绍个性化搜索的过程中，我们会引入推荐（recommendation）的概念。我们甚至可以在用户进行搜索之前就为他们提供个性化的内容推荐。除此以外，大家还会看到，搜索引擎可以成为一个构建推荐系统的强有力的平台。图 11.1 展示了一个由相关性技术工程师实现的、与搜索一起工作的推荐功能。

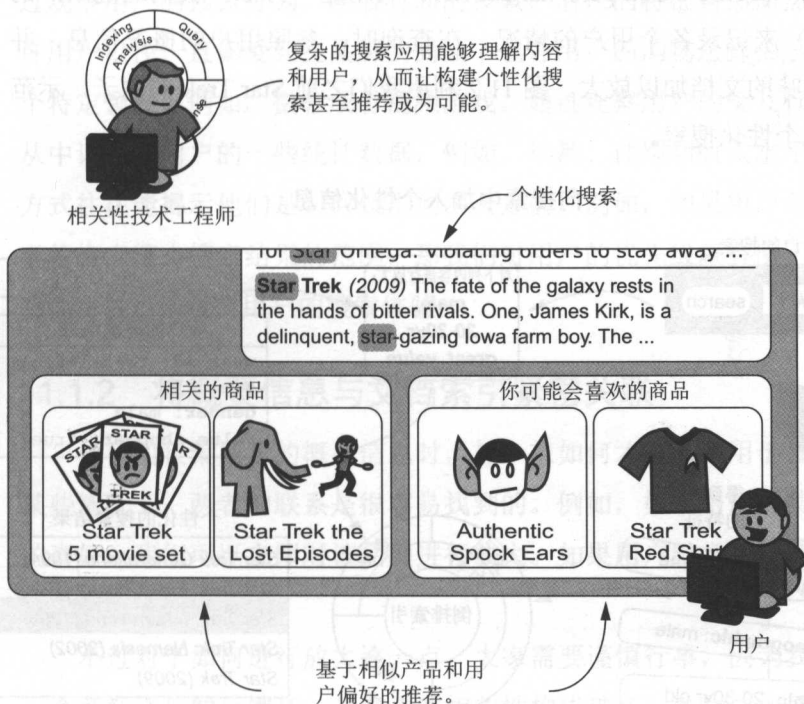


图 11.1 通过结合对内容和用户的认知，我们可以将搜索扩展到其他任务，比如个性化搜索和推荐功能。

## 11.1 基于用户概况的个性化搜索

到目前为止，我们已经根据搜索结果与用户当前的信息需求相匹配的情况，为相关性进行了定义。但随着时间的推移，当对用户的了解愈加深入之后，我们就应该能将他们的口味和偏好都纳入到搜索应用中。这种做法被称为个性化搜索。

我们已经强调过了，在本书中，搜索引擎的核心是一个复杂的 token 匹配与文档排名系统。为了确保搜索对文档的匹配和排名能够反映我们对相关性的认识，本书之前曾经讨论过一系列技术。现在谈到了个性化搜索，搜索引擎的这些基本特性并没有改变。并不存在什么特别的神秘功能或者隐藏特性来实现个性化搜索。搜索要做的仍然是设计有价值的信号，并利用搜索引擎，通过查询和分析对信号进行建模。主要的不同之处在于，这里并不是只从文档中提取信息，我们还会将用户本身看作一种新的信息来源。

了解了这些信息之后，我们把注意力转向构建个性化搜索的第一种方法：基于用户概况的个性化（profile-based personalization）。利用这一方法，我们通过用户的概况信息（profile）来记录各个用户的情况。在查询时，参照用户的概况信息，并据此对满足用户口味的文档加以放大。图 11.2 利用我们之前 Star Trek 的例子，示范了基于用户概况的个性化搜索。

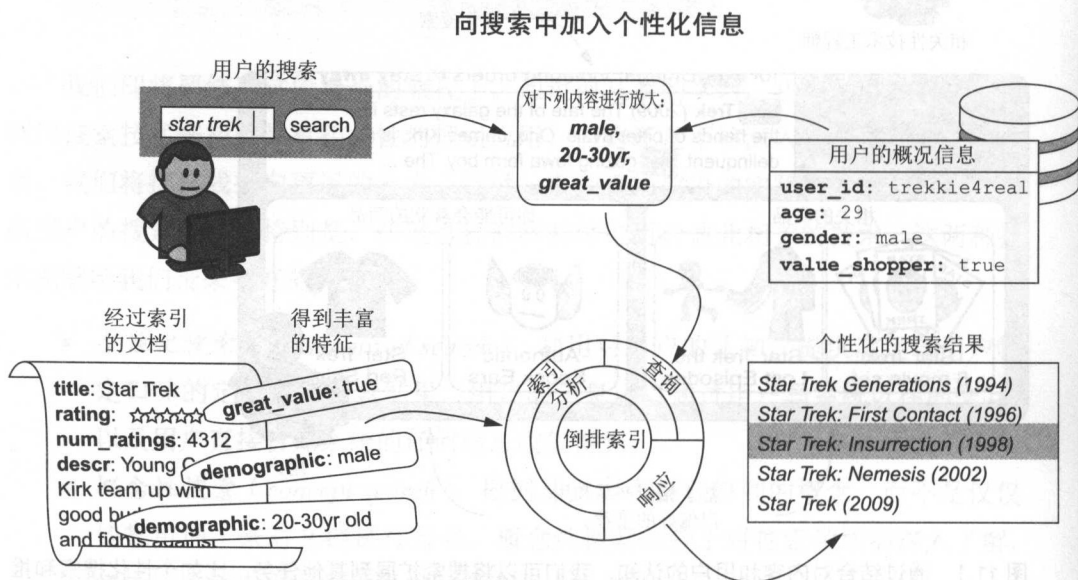


图 11.2 利用用户的概况数据，包含统计数据和偏好信息，为搜索增添个性化。

### 11.1.1 收集用户的概况信息

可是，我们如何为用户的概况数据收集信息呢？关于这一点，如果我们足够幸运，拥有一个积极参与的用户群体，那就不妨建立一个概况信息的页面，等待用户来告诉我们关于他们自己的信息。我们要确保能够为用户填写概况信息提供一定的激励。对于面向社交的站点，我们可以将概况信息公开。这样，用户自己就可以通过概况信息来反映他们的个性诉求了：允许用户以自由文本的形式来描述本人信息。以他们感兴趣的分类为自己的概况信息添加标记。对于非公开的概况信息，要让用户懂得，建立概况数据可以为自己带来更加个性化的使用体验。例如，我们可以直接向用户询问其偏好，并指明这些信息将对应用的行为产生影响。我们可以通过开放各种功能来激励用户去创建概况信息；例如，可以让用户为他们喜欢的内容添加书签，或者将内容分享给好友。

如果缺少概况信息页面，我们仍然可以通过与用户的交互来收集概况信息。通过观察用户的搜索行为，随着时间的积累，潜在的特征自然而然就会显现出来。也许用户以往一直偏爱某些品牌的商品；也许用户的购物选择揭示了其所感兴趣的某个特定领域，例如，摄影或者视频游戏。通过观察用户的交互行为，也许我们可以从中识别出用户的一些统计数据，例如，年龄、性别和收入水平。用户过滤搜索的方式往往能揭示他们是如何做出购买决定的。例如，如果用户是通过产品的评论或者价格来缩小搜索结果的范围，那我们对用户的优先级就有所了解。所有这些信息都可以用于优化用户的搜索体验。

### 11.1.2 将概要信息与文档索引紧密关联

当我们收集用户的概要信息时，想一想如何才能将其用于搜索的解决方案。在某些情况下，两者的联系是很容易找到的。例如，如果用户对某一品牌显示出特别的喜好，我们可以立即对该品牌进行放大。如果用户经常查看评论，那就对具有好评的商品进行放大。

不过对于如何进行放大这一点，大家需要谨慎行事，因为我们有可能会建立一个意料之外的反馈环，对搜索的相关性构成破坏。例如，如果用户不止一次购买了 Acme 公司的产品，也许我们就应该对该用户的搜索结果中出现 Acme 的情况进

行放大。但是如果放大处理过度，搜索页面就可能会完全被 Acme 产品所淹没——相关性较低的 Acme 产品也会出现在那里。在后续的搜索中，用户在行为上会体现出与 Acme 产品更多的交互，这并不是因为他们喜欢这些商品，而是因为我们的放大处理，使 Acme 产品的出现较之其他产品更加普遍。更为糟糕的情况是，这些交互可能会被看作用户对 Acme 产品的偏爱程度在增加，从而导致 Acme 的产品被进一步放大。随着 Acme 产品的“泛滥成灾”，我们很快就可能会发现，顾客根本不再使用我们的搜索应用了。因此，最好的办法是，只有当我们非常肯定用户对特定的产品分类有偏爱时，才对这些产品分类进行放大——比如确实发生了购买行为，而不只是浏览商品页面。

有时，为了匹配来自用户概要数据的信息，我们需要在索引中添加新的信号。如果知道用户偏爱物美价廉的产品，我们就不能简单地对所有低于 20 美元的产品进行放大。一个 20 美元的搅拌机性价比很高，但是一罐 20 美元的豆子就太贵了。相反，我们应该将文档和用户与某种普遍的价值评价标尺（从“廉价品”到“高档品”）联系起来。

像年龄、性别这样的客户群体统计信息可能是搜索的另一种很好的信息来源。假设某位用户的概要信息表明该用户是一名年轻男士。如果我们知道哪些商品在这一群体中卖得更好，那就可以在搜索结果中对这些商品进行放大。为了实现这一功能，我们可以在索引过的文档中加入一个新的字段，列出高度关联相应产品的客户群体。为该字段标注客户群体统计数据的任务可能就要落到内容管理员的肩上了，而信息本身则可能会来自于市场研究。

有了大量业务数据的积累之后，客户群体统计数据的另一个来源就是我们的搜索日志，其对各类不同客户群体的交易记录进行统计。下次对文档重新进行索引时，我们就可以将这些信息加入代表客户群体偏好的字段中了。一旦这些数据进入索引，个性化搜索就会变得非常容易，就像和使用当前用户的统计数据进行放大一样容易。

## 11.2 基于用户行为的个性化搜索

在前一节里，我们向大家展示了，对用户的了解可以通过观察用户在应用中的行为来达成。在这一节里，我们将利用协同过滤算法（collaborative filtering）将这

个概念发挥到极致。这项技术利用以往用户与商品之间的交互信息（浏览、评分、购买等），找出可以天然归属为一类的商品。例如，协同过滤提供了一种算法，可以得出“购买过芭比娃娃的用户可能也会对女式童装感兴趣”这样的结论。我们可以将这一信息纳入搜索中，以提供更加个性化的搜索体验。我们称之为基于行为的个性化。本节，我们将讨论一个有关协同过滤的简单例子，看一看我们是如何将其纳入搜索的。

### 11.2.1 引入协同过滤

对基于行为的个性化搜索而言，我们将目标做了限定。不考虑用户的统计信息、搜索历史和概况数据，我们只关注用户对商品的交互行为。在这一节里，我们将专门考察用户对商品的购买行为。原则上，任何行为都可以称为交互，比如：商品浏览、编辑保存、评分、分享等。给定一组用户对商品的交互行为数据，我们可以利用协同过滤来揭示出隐藏在背后的用户与商品之间的关联关系。

协同过滤有多种形式，从简单的基于计数的方法（counting-based methods，我们马上就会介绍）到非常复杂的矩阵分解技术（matrix decomposition techniques，不在本书讨论范围之内）。但是无论使用哪种技术，协同过滤的输入和输出都遵循同样的模式。

如图 11.3 所示，此处的输入是一个矩阵，代表了位于索引中的用户与商品的交互行为。其中，每一行对应一位用户，每一列对应一件商品。矩阵中的数值表示用户的交互行为。在最简单的情况下，矩阵的值代表一次交互是否发生。例如，用户浏览或购买了某件商品。而在大多数情况下，矩阵的值则可以表示用户对商品的交互是积极的还是消极的。例如，矩阵的值可以表示用户对过去购买过的商品所给的评分。

协同过滤的输出是一个模型，从中可以找出哪些商品与给定的用户或商品关联最为紧密。因此，给定一个源商品（source item），比如苹果，协同过滤模型可能会返回一系列与苹果高度关联的商品，比如香蕉、橘子或者葡萄。而且，返回的每一件商品还包括一个代表关联密切程度的分值（affinity score）。假设输出的分值为香蕉：132，橘子：32，葡萄：11。那么在这里，相对而言，香蕉与苹果的关联最为密切，而葡萄则最为疏远。



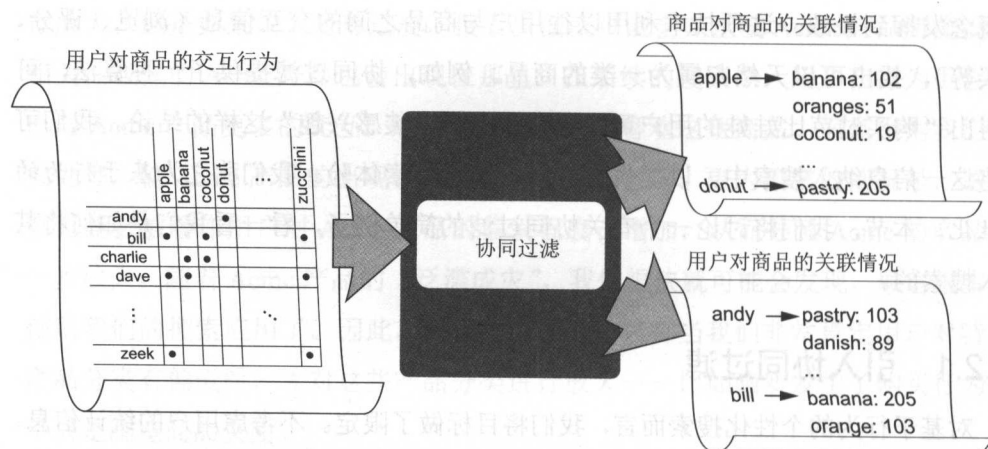


图 11.3 不论采用什么方法,协同过滤的主要工作是接受一个用户对商品的矩阵,并返回一个模型,以快速找出用户对商品或商品对商品的关联情况。

### 11.2.2 使用共现计数的基本协同过滤算法

为了能更好地理解协同过滤是如何工作的,让我们一起来看一个使用共现计数(co-occurrence counting)方法的简单例子。下面的算法过于简单,我们将它作为入门介绍之用,并不推荐在生产系统中使用。尽管如此,它为我们建立起了对协同过滤的基本认识,消除了对协同过滤或多或少的神秘感。正如我们所知道的那样,许多机器学习的算法都是立足于一些简单想法的,比如统计商品被一起购买的次数。

来看一个例子,假定我们为一家电子商务网站工作,手里有一份所有用户购买的所有商品的日志记录。表 11.1 给出了一个示例。

表 11.1 以表格形式记录用户的购买历史

日期	用户	商品
2015-01-24 15:01:29	Allison	Tunisia Sadie 裙装
2015-01-26 05:13:58	Christina	Gordon Monk 高跟鞋
2015-02-18 10:28:37	David	Ravelli 铝制三脚架
2015-03-17 14:29:23	Frank	Nikon 数码相机
2015-03-26 18:11:01	Christina	Georgette 女士上衣
2015-04-06 21:50:18	David	Canon 24 mm 镜头



续表

日期	用户	商品
2015-04-15 10:21:44	Frank	Canon 24 mm 镜头
2015-04-15 21:53:25	Brenda	Tunisia Sadie 裙装
2015-07-26 08:08:25	Elise	Nikon 数码相机
2015-08-25 20:29:44	Elise	Georgette 女士上衣
2015-09-18 06:40:11	Allison	Georgette 女士上衣
2015-10-15 17:29:32	Brenda	Gordon Monk 高跟鞋
2015-12-15 18:51:19	David	Nikon 数码相机
2015-12-20 22:07:16	Elise	Ravelli 铝制三脚架

我们要做的第一件事情就是按照用户对这些购买记录进行分组，如表 11.2 所示。

表 11.2 决定商品共现关系的第一步是按照用户对商品进行分组，一个点 (•) 表示一次购买记录

	Tunisia Sadie 裙装	Gordon Monk 高跟鞋	Georgette 女式上衣	Nikon 数码相机	Canon 24mm 镜头	Ravelli 铝制三脚架
Allison	•		•			
Brenda	•					
Christina		•	•			
David				•	•	•
Elise			•	•		•
Frank				•	•	

下一步就是见证一切“奇迹”的时刻了。对于任何给定的商品 A，我们对同一用户购买商品 A 的同时也购买商品 B 的次数进行统计（此处的术语“同时”，co-occurs，并不是指购买行为同时发生的意思，而是指购买行为是由同一用户发起的）。我们为出现在购买历史记录中的每一对商品都执行该计算。当统计完所有“共现”情况出现的次数之后，我们就得到了任意商品 A 和商品 B 之间的一种关联关系的度量结果。

根据表 11.2 所给的信息，我们来举一个有针对性的例子，考虑一下 Canon 24mm 镜头和索引中其他商品的关联关系。大家会发现，只有一位用户，David，同时购买了 Canon 24mm 镜头和 Ravelli 三脚架；因此，这两种商品的共现次数为 1。但是

有两位用户，David 和 Frank，同时购买了 Canon 镜头和 Nikon 相机。这一对商品的共现次数就是 2。最后，没有用户同时购买 Canon 镜头和 Tunisia Sadie 裙装。因此，它们的共现次数为 0。当对索引中出现的每一对商品都执行这一运算之后，我们所得到的度量结果如表 11.3 所示。

表 11.3 购买历史中出现的每一种商品的共现次数统计

	Tunisia Sadie 裙装	Gordon Monk 高跟鞋	Georgette 女式上衣	Nikon 数码相机	Canon 24mm 镜头	Ravelli 铝制三脚架
Tunisia Sadie 裙装	-	1	1	0	0	0
Gordon Monk 高跟鞋	1	-	1	0	0	0
Georgette 女士上衣	1	1	-	1	0	1
Nikon 数码相机	0	0	1	-	2	2
Canon 24mm 镜头	0	0	0	2	-	1
Ravelli 铝制三角架	0	0	1	2	1	-

这些值表明了每一对商品彼此间关联关系的强弱程度。请注意，正如我们所期望的那样，本例中时尚类商品和其他时尚类商品有着更高的共现关系。与之类似，摄影类商品与其他摄影类商品也有着更高的共现关系。在有些情况下，时尚类商品和摄影类商品的购买行为也存在共现的情况。这也是预料之中的事，因为有一小部分用户对时装和摄影都有兴趣。

商品与商品的关联性可以被直接用于进行基于商品的推荐。表 11.3 中所列的数据可以被存入一个键值型（key-value）存储结构中。于是，当用户访问了 Ravelli 铝制三脚架的详情页面时，系统就会到键值型存储结构中去查找该商品，返回与之对应的一个具有高度关联的有序商品集合（本例中即 Nikon 数码相机和 Canon 24mm 镜头），并把这些商品作为推荐呈现给用户。如图 11.4 所示，亚马逊在为我们返回商品推荐的时候就是这么做的。

进一步分析，我们还会发现用户与各品类商品之间的关联。为此，请参照表 11.2 中列出的用户对商品的购买记录，针对用户发起的每一次购买行为，收集相应的商品对商品的关联情况，并将其汇总相加。例如，Allison 购买了 Tunisia Sadie 的裙装和 Georgette 的上衣。表 11.4 给出了根据共现度量统计得到的对应行，以及这些行的合计汇总。

购买了此商品的客户还购买了

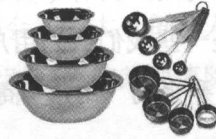
Page 16 of 25



Hamilton Beach 58149  
Blender and Chopper  
★★★★☆ 2,741  
\$26.49 Prime



Crock-Pot SCCPVL600S  
Cook' N Carry 6-Quart Oval  
Manual Portable Slow  
Cooker, Stainless Steel  
★★★★☆ 1,409  
\$24.00 Prime



Kitchen MissionTM  
Stainless Steel Mixing  
Bowls 1.5, 3, 4, and 5 quart.  
Plus Measuring Cup and...  
★★★★☆ 146  
\$21.99 Prime



Cuisinart CTG-00-CCR7  
Curve Crock with Tools,  
Set of 7  
★★★★☆ 136  
\$25.49 Prime

图 11.4 商品对商品的关联可用来生成“相关商品”的推荐。当用户打开 Frigidaire 微波炉的页面时，我们也可以与亚马逊提供的推荐类似，将与微波炉具有高度关联的商品显示在一个特定的区域内。

表 11.4 将反映某位用户购买行为的、商品对商品的度量结果逐行相加，就可以得到用户对商品的关联情况

	Tunisia Sadie 裙装	Gordon Monk 高跟鞋	Georgette 女式上衣	Nikon 数码相机	Canon 24mm 镜头	Ravelli 铝制三脚架
Allison 购买了 Tunisia Sadie 裙装	-	1	1	0	0	0
Allison 购买了 Georgette 女式上衣	1	1	-	1	0	1
合计	1	2	1	1	0	1

当为每一位我们所关注的用户做完相应的汇总统计之后，我们就得到了一个如表 11.5 所示的矩阵。这些值代表了每一位用户与相应品类中的每一件商品之间的关联情况。

表 11.5 用户对商品的完整关联矩阵

	Tunisia Sadie 裙装	Gordon Monk 高跟鞋	Georgette 女式上衣	Nikon 数码相机	Canon 24mm 镜头	Ravelli 铝制三脚架
Allison	1	2	1	1	0	1
Brenda	1	1	2	0	0	0
Christina	2	1	1	1	0	1
David	0	0	2	4	3	3
Elise	1	1	2	3	3	3
Frank	0	0	1	2	2	3

因为我们是从对商品的购买开始统计的，追踪用户与已购商品之间的关联通常而言意义不大。追踪与用户具有零关联的商品数据对推荐而言也没有什么意义；为什么给用户推荐的是一些我们认为用户不会感兴趣的東西呢？因此，让我们去掉这些数据，然后看一下剩余的“用户 - 商品”关联数据，如表 11.6 所示。

表 11.6 用户对已购商品的关联矩阵，去掉了零关联商品

	Tunisia Sadie 裙装	Gordon Monk 高跟鞋	Georgette 女式上衣	Nikon 数码相机	Canon 24mm 镜头	Ravelli 铝制三脚架
Allison	-	2	-	1	-	1
Brenda	-	-	2	-	-	-
Christina	2	-	-	1	-	1
David	-	-	2	-	-	-
Elise	1	1	-	-	3	-
Frank	-	-	1	-	-	3

去掉干扰数据之后，协同过滤所起的作用很容易就显现出来了。和前面的商品对商品的情况一样，这些信息可以直接被用于个性化推荐。如果有办法能将这些信息纳入我们的搜索应用，那就太好了！不必担心，很快我们就可以看到了。

查看数据，我们会发现时尚类商品的购买者与时尚类商品之间有很高的关联性，而摄影类商品的购买者则与摄影类商品之间有着很高的关联性。但是因为其中有一位用户，Elise，对摄影和时尚都有兴趣，所以就产生了跨越两者的交叉推荐。由于这一原因，当 David 看到系统向他推荐 Georgette 的女式衬衫时，可能会感到很奇怪。不过所幸的是，随着收集的数据越来越丰富（包括更多的商品种类，以及每一类商品中更多的购买记录），类似这样的交叉推荐将会变得越来越少，用户对商品的关联从统计学角度出发将主要由大量出现的共现情况来决定。

而且，在更大规模的数据集中，如果像这样不太常见的交叉情况的确存在，这常常也是一种幸运，因为它们可能代表了各品类商品间的潜在关系。例如，Mentos 和 Diet Coke 两者之间唯一存在共性的地方就是它们在某种程度上都属于食品。但是到了 2005 年年底，当 Mentos+Diet Coke 的实验在互联网上开始风靡时，同时购买这两种商品的现象就很有可能会出现。这一现象强调了一个事实，那就是协同过滤可以识别出一些潜在的关联关系，这些关系只通过查看文档的字面内容是无法得到的。

正如我们前面所讲到的，用这种方法来寻找关联性是极其简陋的。例如，对于

那些非常普遍的大众商品，我们并没有做归一化处理，比如袜子。不论我们对时尚、摄影，或者任何其他能想到的东西是否感兴趣，大家总是会经常购买袜子的。因此，袜子与索引中任何一件商品之间的共现次数都会非常大；每个人都会收到有关袜子的推荐。为了解决这一问题，我们需要将每一对商品的共现值除以一个代表普遍程度的量。

基于共现的协同过滤方法并不是生成商品对商品或用户对商品关联性的唯一方法。如果我们正在考虑构建自己的推荐方案，最好去考察一下各种以矩阵为基础的协同过滤方法，例如：截断奇异值分解（truncated singular-value decomposition）、非负矩阵分解（non-negative matrix factorization）和交替最小二乘法（alternating least squares，因 Netflix 公司的影片推荐挑战而闻名）。这些方法都不如我们在这里给出的简单共现计数法那么直观，而且实现它们也更具挑战性。但是，这些方法常常能给我们带来更好的结果，这是因为它们对商品与用户之间的关系有着更加全面的理解。要深入研究推荐系统，我们向大家推荐 Kim Falk 的 *Practical Recommender Systems* 一书（Manning 出版社，2016 年）。不论选择哪种方法，要记住的是，最终得到的结果是一个可以让我们快速找到商品对商品或用户对商品关联关系的模型。这一理解对于我们解释如何将协同过滤的结果用于搜索上下文是非常重要的。

### 11.2.3 将用户行为信息与文档索引紧密关联

在前一节里，我们向大家示范了如何构建一个简单的推荐系统。但我们要讲的应该是个性化搜索啊！本节，我们重新回到搜索这一话题，为大家介绍如何运用协同过滤的输出结果来营造更加个性化的搜索体验。与此同时，我们也会指出一些需要注意的地方。

我们有多种方法将协同过滤得到的信息纳入搜索之中。此处给出的三种策略彼此具有相关性，这是因为它们都是标准的纯文本搜索，且都是结合了协同过滤的乘法放大。其工作原理是这样的：考虑一个基准查询的例子，其中用户的查询为“Summer Dress”，同时针对两个字段进行搜索，分别为 title 字段和 description 字段，如清单 11.1 所示。



### 清单 11.1 基准查询

```
{ "query": {  
  "multi_match": {  
    "query": "summer dress",  
    "fields": ["title^3", "description"]}}}
```

针对该基准查询，我们通过基于 `function_score` 查询的乘法放大，将协同过滤纳入进来，如清单 11.2 所示。

### 清单 11.2 可以利用乘法放大来接纳协同过滤

```
{ "query": {  
  "function_score": {  
    "query": { "multi_match": {  
      "query": "summer dress",  
      "fields": ["title^3", "description"]}},  
    "functions": [{  
      "filter": { COLLAB_FILTER },  
      "weight": 1.1}}]}
```

在这个简单的实现中，通过协同过滤被放大的文档将由 `COLLAB_FILTER` 过滤器的输出内容来决定（我们将稍后讨论）。值得注意的是，该过滤器并不是对结果集中的任何文档都进行过滤。相反，匹配此过滤器的文档都会得到一个 1.1 倍的乘法放大，该数值由权重参数来指定。清单 11.2 中的查询所返回的文档与清单 11.1 中的查询所返回的是完全相同的。但是这其中，任何一篇同时还匹配了 `COLLAB_FILTER` 的文档，在基准查询的评价基础之上，还将获得 10% 的放大。这种放大对搜索结果的排序产生了微妙的影响，于是用户就会发现查询的结果与他们之前的行为变得更加一致了。这就是个性化搜索要实现的目标。

### 查询阶段的个性化

有了基本的框架，我们就可以讨论在搜索时结合协同过滤的三种策略了，这其中的每一种策略都对应了不同的 `COLLAB_FILTER` 过滤器和索引策略。目前，我们假定协同过滤过程的输出结果是一组用户对商品的关联关系——比如“Elise 喜欢 Tunisia Sadie 的裙子、Gordon Monk 的高跟鞋，以及 Canon 的 24 mm 镜头”。但是由于我们是在与机器对话，因此 Elise 对应于 `user381634`，Tunisia Sadie 的裙子对应于 `item4816`，Gordon Monk 的高跟鞋对应于 `item3326`，而 Canon 的 24 mm 镜



头则对应于 item9432。而且我们还假定了，对所有用户和所有品类的产品，我们所掌握的信息是相似的。

基于此数据集，在搜索中结合协同过滤的最为简单直接的方法就是：将协同过滤的数据存入一个键值类型的存储结构中（位于搜索引擎之外），并以此为出发点。我们假设 Elise，即 user381634，有一系列与之紧密关联的商品：item4816、item3326 和 item9432。于是，当 Elise 再次使用搜索引擎时，第一步要做的就是从数据存储中取得与之高度关联的商品，然后再参照清单 11.2，将 COLLAB\_FILTER 替换成另一种过滤器，可以根据 ID 直接对与之高度关联的商品进行放大。

```
COLLAB_FILTER = {  
  "terms": {  
    "id": ["item4816", "item3326", "item9432"]  
  }  
}
```

于是在搜索结果中，任何商品，如果匹配到与 Elise 高度关联的商品，那它们都会被排到最前面。

尽管这是最为明显的方法，但是在查询阶段，这样的方法有可能会非常耗费计算资源。在前面的例子中，与 Elise 高度关联的只有三件商品。在一个更为现实的应用中，与一位用户高度关联的商品，其数量可能会达到几百个或者上千个。而且当数量达到一定程度时，像之前那样将如此多的词（term）放在一起进行“或”运算，会使查询变得非常缓慢。不过，对于这种方法所能支持的规模化扩展的能力，大家也许会感到吃惊。比如，Lucene 在地理搜索方面做得非常不错。但是正如我们在第 4 章里讨论的那样，地理搜索在底层是通过把许多（可能有数百个之多）代表地理区域的词放到一起进行“或”运算来实现的。另外，在许多个性化搜索应用中，不管怎样高度关联的商品数量也不需要上千个那么多；用户搜索的兴趣范围往往是相对较小的。在这一领域里，几百个高度关联的商品也许就可以为用户提供非常显著的个性化搜索体验了。

### 索引阶段的个性化

如果我们的应用无法承载查询阶段的性能开销，那另一种方法就是将性能压力放到索引阶段。这项技术的一个好处在于不需键值类型的外部储存，因为我们会将

协同过滤的信息直接存入索引。

为了实现这一方法，我们在接受索引的文档中增加了一个新的字段，取名为 `users_who_might_like`。顾名思义，该字段包含了可能喜欢给定商品的所有用户的一个列表。例如，当对 Gordon Monk 高跟鞋进行索引时，我们将搜索需要的所有典型信息都包含了进来，如：名称、描述、价格等。不过这一次，我们还包含了一个 `users_who_might_like` 字段，代表与该商品高度关联的所有用户的一个列表。对照表 11.6，我们会发现 Allison (`user121212`) 和 Elise (`user989898`) 都与这个商品高度关联。在这种情况下，`users_who_might_like` 字段的取值将会是序列：`user121212, user989898`。

当所有文档与其对应的 `users_who_might_like` 字段一起被索引之后，剩下的工作就比较容易了。在查询阶段，当 Allison (`user121212`) 进行搜索时，我们发起的查询会带上一个简单的放大过滤器：

```
COLLAB_FILTER = {  
  "term": {  
    "users_who_might_like": "user121212"  
  }  
}
```

同样，这次返回的结果与基准查询得到的结果是一样的，但是其中任何一个文档，如果符合 Allison 作为一名用户“可能对该商品感兴趣”的情况，就会得到一个 10% 的放大——将其排到搜索结果的前面。大家会发现，搜索引擎在查询阶段所执行的这条查询，相比以前更加简单了。这是因为它只包含了一个额外的搜索词 (`term`)。但是使用这一方法，我们必须要注意索引的大小。作为一个参考标准，对于 50 万个篇幅在一页左右的英文文档，这样一个大小适中的规模而言，在其所对应的 Lucene 索引中，我们预计大概有 100 万个全局唯一的词。利用这一方法，每个唯一的用户 ID 都代表了索引中的另一个词。因此，我们应该能够预见到对于数十万的用户量，该方法还是可以应付自如的。但是如果有数百万的用户量，那我们的索引也许很快就会超出服务器所能承受的负载。不过所幸的是，这种方法可以很好地进行水平扩展。我们可以为用户建立代表不同客户群体的分片。如果有百万级的用户量，也许就该为水平扩展分配相应的资源了。

### 索引和查询阶段的个性化

我们要讲的最后一种方法对前面两种方法做了折中。之前，我们用的是用户对商品的关联，但是在这里，我们假设协同过滤的输出结果是如表 11.3 中所示的一组商品对商品的关联。在这个新的方法中，搜索引擎会在查询阶段自动计算用户对商品的关联。

这种方法的准备工作比其他两种方法要更复杂一些。我们将再次需要一个键值结构的存储，用来查询用户的相关信息。但是这次我们存的并不是用户对商品的关联，而是用户最近的购买记录。我们还在索引中增加了一个叫 `related_items` 的新字段。顾名思义，这个字段将包含一个具有高度关联的商品 ID 列表。当 Frank 在搜索的时候，我们首先在查询阶段获得了他的最近购买记录——一个 Nikon 数码相机 (`item1234`) 和一个 Canon 24 mm 镜头 (`item9432`)——然后执行带有下列放大过滤器的查询。

```
COLLAB_FILTER = {  
  "terms": {  
    "related_items": ["item1234", "item9432"]  
  }  
}
```

我们的查询使用了一个 ID 列表，就像前面介绍过的第一种方法那样，但是列表长度更短。同时和第二种方法一样，我们必须对一个包含 ID 列表的额外字段进行索引，但是除非我们有数以百万计的商品，否则这种方法所要处理的信息也要比前面的方法少得多。

正如我们在本节开始处提到的那样，前面介绍的方法不过是将协同过滤纳入搜索的众多方法中的少数几种。我们可以通过很多方式来改进这些方法。例如，大家可能注意到了，在这些方法中没有任何一种方法提到了关联值 (`affinity values`)。相反，我们将所有商品分成了两组：高度关联的商品（匹配 `COLLAB_FILTER` 过滤器）和关联度较低的商品。我们可以对这些方法进行修改，将个体的关联值考虑在内，但是这需要使用一些创新性的技术，包括定义载荷 (`payloads`) 和编写脚本 (`scripting`)，或者甚至有可能包括编写自定义的搜索引擎插件。

### 11.3 构建概念性搜索的基本方法

除了前几章中介绍过的相对标准的方法以外，个性化搜索不过是探索的众多可能的方向之一。搜索应用的另一个有趣的扩展是概念性搜索（concept search）。在阅读本书之前，大家可能认为，搜索就是寻找与用户提供的关键词和过滤条件相匹配的文档这样一个过程。现在，希望大家能够逐渐意识到，一个好的搜索应用是可以推断出用户意图的，并且其所提供的文档会携带用户想要寻找的信息。而概念性搜索则将这一认识发挥到了极致。

概念性搜索的目标是要增强搜索应用的功能，从而使搜索应用在某种意义上理解用户查询的意图。基于这样的理解，通过概念性搜索返回得到的文档也许并不匹配用户的任何搜索关键词，但是却依然包含用户正在寻找的有意义的信息。借用 Google 的一句话，概念性搜索的目标是允许用户“搜索内容，而非文字（Search for things, not strings）”。

或许一个例子可以帮助大家理解我们对概念性搜索的需求。假设有一个关于医学杂志的搜索应用。如果我们使用典型的、基于字符串的方法，搜索“Heart Attack”往往得不到理想的结果。这是为什么呢？因为医学文献中会使用各种各样的术语来描述 heart attack（心脏疾病），例如“myocardial infarction（心肌梗塞）”“cardiac arrest（心脏骤停）”“coronary thrombosis（冠状动脉血栓形成）”等。有大量关于心脏疾病的文章压根就不会提到“heart attacks”。概念性搜索会返回有关心脏疾病的文档，甚至包括那些内容中恰好没有出现该短语的文档，从而为用户提供更好的搜索体验。

还是那句话，我们再怎么强调也不为过——搜索引擎的核心是一个复杂的 token 匹配与文档评价系统。概念性搜索的核心并没有那么神秘；它利用的是可以提高搜索查全率的新的相关性信号，从而扩大了查询和文档的范围。通过仔细平衡这些新的概念性信号（concept signal），我们可以保证搜索结果的高准确性。在本节中，我们将讨论几种人为驱动的方法（human-driven methods），增强我们的搜索应用，使其表现出更多概念性搜索的特征。

### 11.3.1 构建概念性信号

首先,我们可以采取人工为文档添加标签(human-powered document tagging)的方式来实现概念性搜索。我们可以建立一个字段,用来回答问题:“这篇文档是关于什么的?”并将该问题的答案(相应的词汇和短语)统统存入其中。该字段将成为我们新的概念性信号(concept signal)的依据来源。

利用这样的方法,当医学杂志应用的用户在搜索“Heart Attack”时,如果漏掉了一篇重要的文章,我们就可以将短语“heart attack”加入概念性字段。这样就可以匹配到这篇文档了。这种做法也有助于对文档的评价进行微调。例如,假设我们有一篇关于心脏疾病的重要文档,甚至可能包含了短语“heart attack”。但遗憾的是,它出现在了搜索结果的第二页。我们没有从全局角度来试图解决这一问题,而是将短语“heart attack”加入了概念性字段(甚至可以添加多次)。这样一来,不管任何时候,只要用户搜索“Heart Attack”,我们就可以将该文档的评价价值略微调高。

不过这里要提醒大家注意的一点是,依靠人工来管理内容可能难度较大并且耗费资源。为文档添加准确的标签需要具备广泛的专业知识并要求严格的前后一致。例如,涉及心脏疾病的查询是否也应该加上“heart”标签呢?在我们这个用户群体里,“acute heart attack”与“heart attack”有区别吗?一篇文档是否应该同时具备这两个标签呢?只有受过专业训练、具备领域知识的内容管理人员才能分辨出这些细粒度的差异。而且添加标签也需要花费大量的人力。也许我们需要对内容进行深入阅读,因此可能很难做到规模化扩展,以应对实际应用中的数据量。

为了减少内容管理的工作量,一种可行的办法是以“众包”(crowd-source)的方式将概念性信号交给用户来处理。大家还记得我们在第9章中讲到的“抖动(thrashing)”现象吗?当发生“抖动”时,用户因为对搜索结果不满意,很快从一种搜索转换到了另一种搜索,这表明搜索结果与他们的期望并不匹配。设想有一位用户在搜索“Myocardial Infarction(心肌梗塞)”,他在返回的结果页面上停留了大约20秒的时间,然后又进行了一次新的搜索,搜的是“Cardiac Arrest(心脏骤停)”。很明显,该用户并没有找到他所期望的内容。

不过,通常这些用户最终都能找到一个具备相关性的搜索结果。一旦找到了他们想要的内容,就好像用户在告诉我们说,“嘿,记住我刚才在搜的所有其他内



容了吗？这就是我想要表达的意思！”假设在我们的例子里，用户在涉及“cardiac arrest”的搜索结果中仍然没有找到任何感兴趣的东西，于是又提交了一次查询——这一回搜的是“Heart Attack”。看到搜索结果之后，用户单击了结果集中的第二篇文档，然后就没有再发起新的搜索了。这位用户以隐含的方式告诉我们，短语“heart attack”“cardiac arrest”和“myocardial infarction”在某种程度上是彼此相关的。因此，为什么我们不利用搜索词的“抖动”现象，将其加入概念字段，用以获得最终能满足用户信息需求的文档呢？这样一来，如果下次再有人和我们这些发生“抖动”现象的用户一样，遇到了同样的情况，那他们就很有可能在搜索返回的第一页结果中找到其所需要的信息。

还是那句话，对于新引入的概念性字段而言，其主要的目标是要增加搜索的查全率。但是我們也不应该忽视它对查准率的影响。在前面的例子中，如果用户最初搜的是“Cardiac Arrest”，但随后却转而去搜“Gall Bladder”了，那我们的概念性信号就有可能变成干扰因素。我们要确保能适当地平衡概念性信号和现有的其他信号。如果概念性字段是由人工维护的，那么和根据用户生成的其他字段相比，这种字段的质量可能会更高，在相关性评价中所体现的影响也应该更大。

### 11.3.2 利用同义词对内容进行扩充

同义词分析（synonym analysis）是将更深层次的概念性理解植入搜索的另一种有效的方法。当打开同义词文件，加入第一个词条时，我们就已经开始在构建概念性搜索了，如下所示：

TV, T.V., television

当用户向搜索应用查询“TV”时，搜索应用的反馈是：“这里有关于 TV 的文档——不过，我估计你可能对其他包含单词 T.V. 和 television 的文档也会感兴趣，是吗？”

一开始，对文档的同义词进行扩充在某种程度上是手工进行的。内容管理者也许会手工建立起一个丰富的同义词列表，代表某个字段的各种定制的专业术语。在某些范围更大的领域里——医学领域又是一个极好的例子——对于像医学主题词表（Medical Subject Headings，简称 MeSH）这样可以公开访问到的分类体系而言，抛开其原先的用途而将之用于同义词分析，这样的可能性还是存在的。



在使用同义词时，有一件事情需要考虑，是否要采用层级结构来对同义词进行编码。例如，在前面那个涉及“television”的简单例子中，所有同义词的词条在语义上都是处于相同水平的；它们的确都是同义词。但是，正如第4章中所讨论的那样，利用同义词在被索引的词（term）中对专指性（specificity）这一概念进行编码，这样的做法是很常见的。例如，请见下面的同义词条：

```
marigold => yellow, bright_color  
canary => yellow, bright_color  
yellow => bright_color
```

此处，我们对颜色为黄色的东西按层级结构进行了编码（大家可以想象一下，针对所有颜色的一个更大规模的同义词集合，将会是什么样的）。如果使用得当，这样的同义词可以帮助用户扩展有限的查询，例如将“Canary（金丝雀）”扩展为一个更为广义的概念：yellow（黄色）。这样做提高了查全率，用户得到的返回结果中，也会包含一部分使用了更具一般性的术语进行描述的文档。

和往常一样，我们必须注意平衡查全率和查准率。所幸的是，在本例中，原生的  $TF \times IDF$  评价值是可以直接拿来用的。也就是说，当使用了同义词分析之后，一些特定的词汇，比如“marigold（金盏花）”，其在索引中出现的频率，将会比“yellow”和“bright\_color”这样的词汇少得多。因此，当用户搜索某个特定的词语时，比如“Marigold”，包含“marigold”和“yellow”的文档都会被返回，只不过包含“marigold”的文档，将会比更为普通的包含“yellow”的文档获得更高的评价价值。

最后需要注意的是，同义词扩充（synonym augmentation）和概念性字段（concept fields）是彼此互补的两种方法。同义词分析通常是将同义词放入原文本所在的同一字段内，将它和原文本一起接受分析。不过，如果我们担心同义词有干扰，那么将其放入一个单独的字段也许不失为一个好主意。这样一来，它们被赋予的权重就可以变得相对低一些了。将概念性字段和同义词扩充结合在一起的另一个好机会是在给文档添加标签的时候。在这种情况下，通过要求人们只提供最具特殊性的标签，我们可以极大地减少人工添加标签的工作量。随后，我们可以利用以层级结构组织而成的同义词，对那些所含标签不具很强特殊性的文档自动进行扩充。

## 11.4 利用机器学习来构建概念性搜索

在前面几节里，我们介绍了利用简单方法来实现的个性化搜索，然后转而讨论更为复杂的机器学习方法。此处我们也遵循同样的方法，从以人工方式对概念进行管理的方法转而讨论我们称之为内容扩充（content augmentation）的机器学习方法。

和之前一模一样，我们的目标是要在文档中包含一个新的反应内容的信号，以此来提高搜索的查全率。尤其是，我们将利用机器学习自动产生“伪内容（pseudo-content）”，并将其追加加入被索引的文档。这些新增的内容并不是人类可读的文本段落。相反，这些“伪内容”将会是一系列不曾在原有文档中出现过的词汇，但从统计的角度出发，在某种意义上，这些词汇又理应存在，因为它们与该文档中出现的概念息息相关。

为了生成新的“伪内容”，我们以包含单词的文档为基础，对单词之间在统计学意义上的关联关系进行了算法建模。例如，假设有一篇医学期刊文章包含了单词“cardiac（心脏病）”。那么，很有可能同一篇文章中也会包含像“heart（心脏）”“arrest（停止）”“surgery（外科手术）”和“circulatory（血液循环）”这样的单词；这些单词都与“cardiac”这一主题相关。但是，同一篇文章中不太可能会出现单词“clown（小丑）”“banana（香蕉）”“pajamas（家居服）”和“Spock（斯波克）”；因为这些单词与“cardiac”主题几乎没有任何共同之处。通过考察单词的共现关系，我们对其彼此间的关联关系逐渐开始有所了解。一旦为这些关联关系建立起了良好的模型，我们就可以针对任意给定的文档，相应产生出一组某种意义上应该在该文档中出现的单词。

让我们来看一个非常简单的例子。假设有一小组文档，如清单 11.3 所示。每一篇文档都是一个明显关于“dog”或“cat”的句子。如果对这些文档进行分析，我们可以提取出 token，对其进行标准化（通过小写转换和词干提取），并过滤出常用的禁用词。最终的结果可以用矩阵的形式呈现出来，如表 11.7 所示。此处，点（•）表示该词（以列表示）在文档（以行表示）中出现了一次或者多次。

清单 11.3 一组形式简单的文档，展示了词汇的共现关系

```
doc1: The dog is happy.  
doc2: A friendly dog is a happy dog.
```

doc3: He is a dog.  
 doc4: Cats are sly.  
 doc5: Fluffy cats are friendly.  
 doc6: The sly cat is sly.

表 11.7 以矩阵形式表示的单词以及包含这些单词的文档。(禁用词得到了过滤, 复数形式的单词经过了词干提取。此外, 为了让按统计来聚类的单词显得更加醒目, 我们对表列的次序做了调整)

	dog	happy	friendly	cat	fluffy	sly
doc1	.	.				
doc2	.	.	.			
doc3	.					
doc4				.		.
doc5			.	.	.	.
doc6				.		.

大家可能注意到了, 这个“词 - 文档 (term-document)”矩阵与表 11.2 中的“用户 - 商品 (user-item)”矩阵很类似。这并不是巧合。根据用户的交互行为来识别具有关联性的商品, 根据文档的共现情况来识别具有关联性的词, 这两个问题几乎是等价的。前面那个有关个性化搜索的例子揭示了时尚类商品和摄影类商品之间内在的聚类特征, 而表 11.7 则揭示了一个有关“dog”类词汇的聚类和一个有关“cat”类词汇的聚类。原则上, 我们甚至可以使用同样的共现统计方法来识别这些高度相关的词汇。但在实际应用中, 我们往往会用一些更为复杂的方法, 比如潜在语义分析 (latent semantic analysis)、隐含狄利克雷分布 (latent Dirichlet allocation), 或者是最近十分流行的 Word2vec 算法。不过, 这些方法已经超出了本书讨论的范围, 它们所产生的模型允许我们为文档提供有关伪内容的“推荐”。这种“推荐”方法与我们在 11.2.2 节中介绍的, 根据用户对商品和商品对商品的关联度进行的商品推荐方法, 大体上是一样的。

当自动生成的“伪内容”与相应的文档一起被索引之后, 查询就可以匹配到一部分原本并没有包含用户搜索词的文档了。尽管如此, 这些文档基于词共现的统计情况 (statistical term co-occurrence), 会与用户所给的关键字有着非常紧密的关联。在前面的例子中, 搜索“Cat”也许会返回一篇讲述顽皮的毛绒动物 (a sly fluffy animal) 的文档, 尽管文档中并没有包含单词“cat”。

### 11.4.1 概念性搜索中短语的重要性

我们还没有讨论概念性搜索中一个非常重要的概念：短语（phrases）。短语所表达的含义通常要比构成该短语的词更具针对性。一个典型的例子就是：软件开发人员（software developer）。软件开发人员不是软件（software），而是开发软件的人。此外，还有很多其他类型的“开发人员”，例如土地开发商（land developer），它与软件开发毫无关系。

因此，进行内容扩充（content augmentation）之前，首先从统计学意义上将存在于文本中的短语识别出来是非常有用的。这些短语可以被添加到“词 - 文档（term-document）”矩阵的相应列中。这样一来，在内容扩充阶段，这些在概念上含义精准的短语就会被纳入新生成的内容中。

搭配提取（collocation extraction）是一种常用技术，可以用来识别具有统计学意义的短语。将文档的内容以  $n$  元语法（ $n$ -grams）的形式进行划分（通常是二元语法形式的双联词结构，即 bigrams）。然后进行统计分析，判断哪个  $n$  联词出现的频率足够高，将其视为在统计学意义上具有足够的影响力。这样的分析方法，往往是一种表现非常出色的统计算法。例如，如果文档集合中单词“developer”出现了 1000 次，并且在 25% 的情况下出现在单词“developer”前面的都是“software”，那么双联词“software developer”就很可能被标记为非常重要的短语。

## 11.5 连接个性化搜索与概念性搜索

正如我们之前所指出的，个性化搜索和概念性搜索之间存在着很强的联系。两者都依赖于构建新的信号来提高查准率和查全率。两者都使用了相似的机器学习方法。但是，两者间的关系其实比这还要深，因为这些方法可以被结合在一起使用，从而进一步提高搜索的相关性。

来看一看冷启动（cold-start）的问题。假设我们正在尝试构建一个基于协同过滤的个性化搜索。当有一个新的商品被加入到我们的品类时，会发生什么情况呢？大家回忆一下，协同过滤的方法是依赖于用户的交互行为的。由于还没有人与新加入的商品有过交互，因此也就不会有个性化的结果产生。我们无法根据并不存在的行为模式来提供新商品的推荐；这就是所谓的“冷启动”问题。

不过，这引出了一个非常重要的问题。我们的确至少还有关于该商品的一部分信息，即：商品的文字资料。我们可以用它来生成个性化推荐吗？答案是肯定的！实现的方法就是将概念性搜索的一些内容纳入个性化搜索的策略中来。利用概念性搜索，我们通过对文本进行广义的、概念性的理解来扩充文档的内容。当我们将概念性搜索加入个性化搜索中时，与之前不同的是，我们必须扩充用户的概况信息，以追踪其所感兴趣的那些概念。在本例中，正如前一节所讲的那样，这里的概念所代表的，是对用户而言具有高度关联的那些重要的单词和短语。

有很多方法可以用来识别对用户而言具有高度关联的内容。其中一种方法就是重新回到机器学习上来，根据用户与文档的交互行为，以及这些文档的文字内容，以某种方式推断出用户对商品的关联程度。不过，也没有必要弄得过于复杂；因为用户会在他们所发起的搜索中，持续不断地为我们提供具有高度关联的术语。如果我们足够幸运，有很多积极参与的用户，我们甚至可以直接向用户询问，他们对什么类型的内容感兴趣。

现在，让我们换个角度思考一下，如何运用协同过滤中所用到的行为信息对概念性搜索进行内容扩充。在 11.2 节中，我们看到了，协同过滤如何能在不同的摄影器材之间建立起一种关联关系，在不同的时尚类商品之间建立起另一种关联关系。这些关联关系都是仅凭用户行为建立起来的；而商品的文字内容则并没有发挥任何作用。然而，正如本例所展示的那样，在行为上彼此相关的商品往往在概念上也是彼此相关的。因此，当与商品关联的文字内容影响力不足时，我们也可以利用行为信息来帮助用户找到他们想要的文档。

## 11.6 推荐是一种广义的搜索

本书通篇所讲的都是搜索的来龙去脉，我们揭开了搜索引擎的面纱，解析了其内部的工作机制，并且为了生成具有高度相关性的搜索应用，给出了各种技术手段。我们还讨论了业务需求，描述了如何改变文化，让搜索相关性成为企业需要解决的核心问题。在本章中，我们指出了一系列方法，为搜索注入了神奇的魔力，使其能理解用户的意图。

但是在这里，在本章即将结束的时候，我们对过去所讲的所有内容提出了一个



全新的质疑，也就是下面这个问题：

也许我们应该要构建的应用并非搜索，而是推荐。

设想一下，在一个以概念为基础的个性化搜索应用中，如果没有显式的搜索，那会发生什么事情呢？如果搜索框是空的，过滤条件也没有被选中，情况又会如何呢？搜索应用还能继续正常工作吗？没错！即使没有用户的直接输入，搜索应用也依然拥有相当数量的上下文信息，可以用来为用户提供丰富的推荐。例如，如果用户浏览了商品的详情页面，那么应用就可以利用我们在本章中讨论的方法，提供相关商品的推荐。如果用户过去与应用有过交互，在做推荐的时候就可以结合考虑用户的行为信息和概念信息，从而使推荐结果充满个性化。因此大家可以看到，即使没有用户的显式查询，系统仍然可以进行推荐。正如我们在接下来的部分要给大家展示的那样，也许将搜索看作推荐的一个子集更有意义。

我们再深入挖掘一下。想一想现实生活中存在的与搜索和推荐相类似的例子。从最坏的角度来看，一个基本的搜索应用可能就像是一个嘴里嚼着口香糖、对工作提不起兴趣的年青店员。我们说，“我需要一件衬衫”，于是店员指了指挂满衬衫的那面墙。那里有上百件衬衫——可想而知，混合了所有的样式、尺寸和价格。由于多到根本无法挑选，因此我们尝试对搜索进行过滤：“嗯，有 M 号的吗？”店员（仍然飞快地嚼着口香糖）抬头瞥了一眼，然后指了指最下面的货架。那里仍然有很多衬衫要选——混合了各种样式和价格，但是我们确实需要一件新衬衫，因此只好走到货架跟前，开始一件一件地找符合自己尺寸的衬衫。

尽管我们提出了这个购买新衬衫的故事，但其实店员早已开始在为客户提供推荐了。只不过推荐的效果不是特别好。这位店员忽略了另一些可以作为上下文线索的个性化和概念性的因素，这些线索可以帮助客户找到他们真正需要的商品。

继续拿这个故事举例，我们将这位嚼着口香糖的店员换成了自己的私人时装顾问。这一次我们走进商店，对时装顾问说，“我需要一件衬衫”，于是时装顾问直接把我们带到了一个货架前，上面摆放的衬衫都是符合我们的尺寸的。时装顾问是一位做过深入研究的专家，她熟知各种风格的服装款式，并且知道如何搭配，能让我们穿出更好的效果。她对我们的个人风格也是了如指掌。时装顾问匆匆看了一眼货架，取出了一些符合条件的衬衫，并将它们摆在我们的面前供我们挑选。然后她抬



起头问道，“哦，您今天想买什么价位的？”——这是一个额外的上下文信息。在得到了我们的回复之后，她将几件价格过高的衬衫取了出来并放回了货架。最后，她会帮着我们一起在剩下的衬衫中逐一进行挑选。

现在，我们已经取得了一定的成效。这位细心周到、知识渊博的时装顾问，不是把我们撇在一边，让我们自己漫无目的地寻找衬衫，而是和我们一起，帮着给出推荐建议，这些推荐既考虑了我们自身的因素，也结合了时尚领域的元素。你知道吗？在帮我们挑选完衬衫以后，时装顾问还将我们带到了收银台旁摆放帽子的货架前，说道，“看一下这顶帽子吧。它非常适合您，而且和您的新衬衫很搭配”。她是对的；一顶非常棒的帽子！这是典型的推荐，因为，即使没有显式地进行搜索，这位时装顾问也可以根据她手头掌握的各种信息，准备为我们提供一定的反馈。

### 11.6.1 用推荐代替搜索

正如前面的故事所展示的那样，推荐可以被认为是一个非常重要且具有一致性的概念——它恰好涵盖了搜索的概念。下面是一个正式一些的定义：

推荐是指根据手头掌握的最佳信息，为用户提供可供选择的最佳商品的能力。

这一定义中最值得注意的部分是单词“信息”。此处，信息有三个方面的来源：用户、品类中的商品，以及推荐所在的当前上下文。

- 用户信息——因为用户与应用存在交互，所以我们可以识别用户的行为模式，并了解他们的兴趣和口味。特别是，对于一些积极参与的用户而言，他们甚至也许很愿意将自己的兴趣直接告诉我们。
- 商品信息——为了能够给出合理的推荐，熟悉各种品类的商品是非常重要的。至少，商品需要具备可供匹配的、有价值的文字内容。商品也需要具备可供放大和过滤的良好的元数据。在更高阶的推荐系统中，我们还应该利用用户的整体行为，用它来为我们提供新的信息，告诉我们分类中的商品是如何彼此关联的。
- 推荐上下文——为了给用户提供可能的最佳推荐，我们必须考虑用户所处的

当前上下文。用户正在浏览一件商品的详情页面吗？那我们就应该为用户提供相关商品的推荐，只要用户还没有决定在本次交易中购买这些商品。用户选好了商品开始准备结算了吗？那我们就给用户推荐一些非常热销的廉价商品吧。我们要向用户发出订单邮件了吗？那我们就给用户提供一些高度个性化的推荐吧，看看能否把他们变成回头客。

大家可能注意到了，我们在前面的讨论中很少提到搜索。这是怎么回事呢？难道，搜索没了吗？恰恰相反！搜索依然存在；它只不过是推荐的另一种可能的上下文。事实上，搜索是非常重要的上下文，因为它表示用户正在确切地告诉我们现在正在寻找什么。每当用户执行一次搜索，我们就有可能得到极为丰富的信息，并由此能给出更为贴心的推荐。再次回到时装顾问的那个例子，搜索就相当于我们告诉顾问，“你知道吗，今天我想买件 Hawaiian 衬衫”。于是顾问推荐了一件衬衫，不仅满足当前的搜索上下文（是 Hawaiian 的），而且还很符合我们现有的个人偏好。

## 11.7 祝愿大家有一个美好的相关性搜索之旅

本书就要结束了。在结束之前，回顾一下我们都讲了哪些内容：

- 第 1 章帮助大家熟悉了一下问题域，以及当我们着手改进搜索的相关性问题时可能会遇到的一系列挑战。
- 第 2 章分别从内部和外部介绍了搜索技术的工作原理，为本书将要讨论的搜索技术奠定了基础。
- 第 3 章介绍了一系列调试工具，它们对于排查各种相关性问题的非常有用。
- 第 4 章介绍了如何对文本进行处理，以从中提取有价值的特征，用于搜索。
- 第 5 章和第 6 章讨论了如何使用文本特征来构建更高级别的相关性信号，以及将这些信号结合在一起的各种方法。
- 第 7 章解释了如何利用函数和放大处理来进一步调节相关性，调整搜索结果，以满足业务目标。
- 第 8 章向我们揭示了相关性不只是参数调优；它也可以帮助用户理解和完善对他们可用的那些信息。

- 第 9 章结合了前面几章学到的经验，为我们提供了一个端到端的相关性案例，并列举了设计相关性搜索应用的系统性方法。
- 第 10 章描述了如何改变组织的文化，使其重视相关性。
- 第 11 章，也就是本章，拓宽了搜索的视野，将个性化搜索、概念性搜索，以及推荐也都纳入了进来。

讨论完所有这些议题之后，相信大家一定会发现——也许已经发现了——每一个搜索应用都有属于它自己的一系列相关性挑战。不过读完本书之后，大家应该会发现自己已经掌握了更多的知识，可以独自面对并克服这些挑战。

## 11.8 本章小结

- 个性化搜索为个体用户提供了定制化的体验，并允许对高度关联的商品进行放大，将其排到搜索结果的前面位置。
- 通过建立起能够追踪用户偏好以及客户群体统计信息的用户概要数据，我们可以构建基本的个性化搜索。请确保这些信息存在于搜索的索引中。
- 利用协同过滤来创建基于用户行为的个性化搜索。
- 概念性搜索不仅为用户提供了与搜索词相匹配的文档，而且还会提供与搜索意图相匹配的文档。
- 为了提高查全率，概念性搜索要求在索引或用户的查询中添加新的内容。为了保证查准率，仔细平衡新引入的概念性信号，这一点是很重要的。
- 结合使用个性化搜索和概念性搜索，两者相辅相成。
- 在没有显式的用户查询时，个性化搜索和概念性搜索实际上就是推荐。
- 将推荐视为一种广义的搜索。

# 附录A

## 直接根据TMDB建立索引

---

从第3章开始，本书采用的例子都来自 TMDB (The Movie Database)。为了便于使用，我们将这些例子预先打包成了一个 `tmdb.json` 文件。不过，我们都是“授人以渔”的坚定信仰者。在本附录中，我们将为大家介绍如何使用最新版本的 TMDB 数据作为例子。

在此需要提醒大家注意的是，我们对 TMDB 并没有控制权，因此随着时间的推移，TMDB 有可能会更改其 API 或者相关的策略，从而导致本附录所讲的内容变得过时。尽管如此，我们还是非常感激 TMDB 允许本书使用它的数据。在此鼓励大家访问 TMDB 的网站 (<http://themoviedb.org>)，并为他们的辛勤工作提供捐助！

本附录将以如下步骤指导大家：

1. 从 TMDB 获取 API key。
2. 编写 Python 代码与 TMDB 进行交互。
3. 通过爬取 TMDB 的高评分电影，获得一个影片列表。
4. 获取每部影片的扩展数据。

我们将建立一个流程：首先根据影片 ID 对 TMDB 进行爬取，然后通过访问一系列 API 服务端，比如 `/movie/<id>` 和 `/movie/<id>/cast`，对每部影片的详细扩展数据进行提取。实现这些功能的代码可以在 GitHub 库中的“appendix A IPython

notebook examples”里找到 (<http://github.com/o19s/relevant-search>)。

## A.1 设置TMDB的key并加载IPython notebook

首先我们要从 TMDB 获取一个 API key。这个 key 给了我们使用 TMDB API 的授权。当访问 API 时，我们将该 key 传给 TMDB 以标识自己的身份。大家可以按照下列网站所给的步骤获得 API key：[www.themoviedb.org/documentation/api](http://www.themoviedb.org/documentation/api)。

有了 API key，我们就可以打开任何一个命令行窗口，准备执行那些 Python 的示例代码。在使用本附录中的 Python 示例代码对 TMDB 进行访问之前，需要先设置一个环境变量 TMDB\_API\_KEY。

在 Mac/Linux 系统中，做如下设置：

```
export TMDB_API_KEY=<API KEY from above>
```

对于 Windows 系统，则在命令行输入如下命令：

```
set TMDB_API_KEY=<API KEY from above>
```

## A.2 针对TMDB API的设置

接下来，我们将执行一系列与 TMDB 交互所需的配置。我们会安装一个 Python 库，将若干系统库导入我们的 Python 代码中，并将 Python 设置为通过 HTTP 进行访问。

这里用到的唯一一个额外的 Python 库是 requests 库。请确认使用 Python 的包安装工具 pip 已经安装了 requests 库。如果没有 Python 的包安装工具，我们应还有 Python 内置的 easy\_install 工具可以使用：

```
easy_install pip  
pip install requests
```

有了 API key 和所需的库，我们就可以开始从 TMDB 下载影片数据了。不过，首先，为了支持与 API 进行交互的功能，让我们来确认一下代码所需的所有设置是否均已就绪。如清单 A.1 所示的这段代码，导入了所需的 Python 模块，获取了 TMDB\_API\_KEY 环境变量，并创建了一个专门用于和 TMDB API 进行通信的 HTTP 会话。

## 清单 A.1 示例代码是如何实现的

```
import requests
import json
import os
import time

tmdb_api_key = os.environ["TMDB_API_KEY"]

tmdb_api = requests.Session()
tmdb_api.params={'api_key': tmdb_api_key}
```

← 确认Python库request已经安装就绪

← 读取TMDB\_API\_KEY环境变量

← 用来与TMDB交互的HTTP会话

这段代码最重要的部分是最后两行。此处我们调用 Python 的 `requests` 库创建了一个会话，并利用 `tmdb_api_key` 为返回的会话配置了一个 `api_key` 参数。既然现在有示例代码在手，那就让我们直接开始调用 TMDB API 吧。

## A.3 用TMDB API爬取数据

接下来，我们的工作就比较有意思了，那就是：从 TMDB 中提取影片数据！如前所述，代码由两部分组成：

- 用于下载影片 ID 列表的代码。
- 针对每个影片 ID 获取额外详情的代码。

首先，我们会构造一个 `movieList` 函数，用来获得代表影片标识符的一个列表。TMDB API 被组织为一系列可以列出影片清单的服务，例如：`/movies/popular`（根据受欢迎程度列出的影片清单），或者 `/movies/top_rated/`（根据评分由高到低列出的影片清单）。在访问每一个服务时，我们会有一个外层循环，用以传入 URL 查询参数：`page`。因为每个请求一次只返回 20 部影片，所以我们需要连续请求更多的页面，以获得足够的影片供后面搜索使用。

为了实现影片搜索，我们首先会在 `movieList` 函数里把希望纳入搜索引擎的影片数据下载下来。为此，该函数通过 `top_rated` 服务，以分页的形式获取最受欢迎的影片 ID，并将每一个 ID 存入 `movieIds` 列表。当完成所有 ID 的存储之后，`movieList` 就会返回 TMDB 影片的一个 ID 列表，如清单 A.2 所示。



## 清单 A.2 从 TMDB 爬取影片数据——movieList

```
def movieList(maxMovies=10000):
    url = 'https://api.themoviedb.org/3/movie/top_rated'
    movieIds = []
    numPages = maxMovies / 20
    for page in range(1, numPages + 1):
        httpResp = tmdb_api.get(url, params={'page': page})
        try:
            if int(httpResp.headers['x-ratelimit-remaining']) < 10:
                time.sleep(3)
        except Exception as e:
            print e
        jsonResponse = json.loads(httpResp.text)
        movies = jsonResponse['results']
        for movie in movies:
            movieIds.append(movie['id'])
    return movieIds
```

针对当前页访问 top\_rated 服务 ①

对于每一页由 top\_rated 所返回的影片……

解析JSON格式的响应结果 ②

……控制对API的访问（TMDB API的访问控制规则可能会有所变动）……

……遍历每一页中的影片，并保存影片ID……

……返回累积得到的影片ID……

在 movieList 中，我们针对 TMDB API ① 发起了一个 GET 请求。然后，对 HTTP 正文部分的文本进行了解析，并将解析得到的 JSON 正文存入了一个 Python 字典 ②。字典中的每一条数据都包含了 HTTP 响应结果的主体部分，并保存了每部影片的信息，以供我们使用。这里我们不必细究响应结果的数据结构，只需简单地从返回的每部影片中 ID 提取出来，加入一个列表即可，其他内容则可以忽略。这一过程在我们以分页形式从 TMDB 获得额外的响应结果时会被反复执行。

现在，我们来获取每部影片的扩展信息。本书所用的提取函数 extract 是从 tmdb.json 文件中提取影片数据的。在这一版本的提取函数中，如清单 A.3 所示，我们将 movieList 返回的影片 ID 列表传入其中。函数 extract 通过逐一访问每部影片的详细信息，获得了有关于影片的更为深入的信息。该函数还访问了第 5 章和后面几章所需的、有关于演职人员的更多额外的细节。最后，extract 以字典的形式返回了累积得到的影片详情，该字典将影片 ID 映射成相应的详细信息。

请注意，我们是以倒序方式展示这段代码的。首先，extract 每次只提取一部影片，然后调用 getCastAndCrew 函数。紧接着，我们看到 getCastAndCrew 是通过访问每部影片的 /credits 服务端来实现的。

## 清单 A.3 从 TMDB 中提取每一部影片——extract

```

.....访问movie/<id>服务以
获得额外的详细信息

def extract(movieIds=[], numMovies=10000):
    movieDict = {}
    for idx, movieId in enumerate(movieIds):
        try:
            httpResp = tmdb_api.get("https://api.themoviedb.org/3/movie/%s"
                                     % movieId, verify=False)
            if int(httpResp.headers['x-ratelimit-remaining']) < 10:
                time.sleep(6)
            movie = json.loads(httpResp.text)
            getCastAndCrew(movieId, movie)
            movieDict[movieId] = movie
        except ConnectionError as e:
            print e
    return movieDict

```

对于这些影片中的每一个影片ID.....

.....控制对API的访问 (TMDB API的访问控制规则可能会有所变动)

.....解析JSON格式的响应结果, 并将其加入影片字典中

.....利用额外的演职人员信息对影片加以充实

.....返回影片字典

在下面的清单 A.4 中, 我们为大家展示了如何访问演职人员的信息。该函数访问了 TMDB 的 /movie/<movieId>/credits 服务端, 并将演职人员的详细信息添加到了影片记录中。

## 清单 A.4 获取演职人员的信息

```

def getCastAndCrew(movieId, movie):
    httpResp = tmdb_api.get("https://api.themoviedb.org/3/movie/%s/credits"
                             % movieId)
    credits = json.loads(httpResp.text)
    crew = credits['crew']
    directors = []
    for crewMember in crew:
        if crewMember['job'] == 'Director':
            directors.append(crewMember)
    movie['cast'] = credits['cast']
    movie['directors'] = directors

```

访问TMDB的credits服务

解析包含演职人员信息的JSON格式的响应结果

对于每一位演职人员, 将其从字典中取出, 并加入directors列表

将演职人员信息保存到movie中

## A.4 在Elasticsearch中为TMDB影片建立索引

一切就绪之后, 我们就可以利用第3章介绍的 `reindex` 函数为所有影片在 Elasticsearch 中建立索引了。回忆一下, `reindex` 假设我们将 Elasticsearch 运行于 `http://localhost:9200`, 即 Elasticsearch 默认的安装位置。不过, 大家也可以访问本书

的 GitHub 库 (<http://github.com/o19s/relevant-search-book>), 了解 Elasticsearch 的其他运行方式。

出于完整性的考虑, 清单 A.5 重新给出了 `reindex` 函数的实现。此处无须再做赘述。需要记住的重点在于, 该函数利用所给的分析器和映射配置对索引进行了删除和重建。

#### 清单 A.5 `reindex` 函数

```
def reindex(analysisSettings={}, mappingSettings={}, movieDict={}):
    settings = {
        "settings": {
            "number_of_shards": 1,
            "index": {
                "analysis": analysisSettings,
            }
        }
    }
    if mappingSettings:
        settings['mappings'] = mappingSettings

    resp = requests.delete("http://localhost:9200/tmdb")
    resp = requests.put("http://localhost:9200/tmdb",
                        data=json.dumps(settings))

    bulkMovies = ""
    for id, movie in movieDict.iteritems():
        addCmd = {"index": {"_index": "tmdb",
                           "_type": "movie",
                           "_id": movie["id"]}}
        bulkMovies += json.dumps(addCmd) + "\n" + json.dumps(movie) + "\n"
    resp = requests.post("http://localhost:9200/_bulk", data=bulkMovies)
```

默认设置

使用给定的分析及映射配置

删除并重建 tmdb 索引

对给定的影片进行批量索引

有了该函数, 我们就可以将 `MovieDict` 传入 `reindex`, 如清单 A.6 所示。

#### 清单 A.6 在 Elasticsearch 中建立索引

```
movieIds = movieList()
movieDict = extract(movieIds)
reindex(movieDict=movieDict)
```

此外, 如果我们想覆盖 `tmdb.json` 文件的内容, 只要对 `movieDict` 以 JSON 格式进行编码并保存成文件就可以了, 如清单 A.7 所示。

#### 清单 A.7 创建 `tmdb.json`

```
with open('tmdb.json', 'w') as f:
    f.write(json.dumps(movieDict))
    f.close()
```

仅此而已！`tmdb.json` 直接反映的是来自 TMDB 的源数据模型。它保存了来自 `/movies/<id>` 服务端的内容，并利用直接取自 `/movies/<id>/credits` 的内容对其进行了扩充。

## 附录B Solr读者指南

---

欢迎你，Solr 读者！本书介绍的相关性搜索知识同样也适用于你的工作。正如我们前面所讲的，Solr 和 Elasticsearch 在底层核心的 Lucene 搜索库基础上都提供了友好的接口。在本附录中，我们将一章一章地为大家着重指出，对于搜索引擎的一些特性的讨论，如何在 Solr 中找到对应的功能。同时我们也会指出两种搜索引擎在处理相关性方面各自的一些优缺点。

需要澄清的是：我们的目标是为大家的学习提供帮助，因此不会将之前的例子再逐个重新实现一遍。本书的讨论如果只针对一种搜索引擎，那就未免太过单一了。因此我们想尽可能多地将 Solr 的一些功能映射到书中的大部分相关性讨论上来。这部分讨论假定大家已经掌握了 Solr 的一些基础知识以及配置方法。对于本附录所讨论的话题，如果大家想进一步了解，推荐大家访问 Solr 的官方参考指南 (<https://cwiki.apache.org/confluence/display/solr/Apache+Solr+Reference+Guide>)，*Solr Start* 在线系列 ([www.solr-start.com](http://www.solr-start.com))，或者翻阅像 Trey Grainger 的 *Solr in Action* (Manning 出版社，2014 年) 这样优秀的书籍。

为了有效组织我们的讨论，并帮助大家掌握本书讲解的知识，我们大致按照章节对本附录做了划分。幸运的是，这样的划分方式正好与搜索引擎相应的功能组成相符。我们的讨论会涵盖第 4 章到第 8 章，因为这几章关注的是与搜索引擎的实际交互。此处略过了第 3 章，因为我们已经在那一章加入了一些帮助大家入门的脚注。我们也略过了第 9 章，因为它用的是前几章里介绍的 Elasticsearch 的功能。如果有的读者不太想关注具体的讨论细节，也可以浏览每一节中的表格，从中了解两种搜

索引引擎之间大致的对应关系。

那么，现在就开始把我们的 Solr 搜索引擎打造成一件相关性搜索的利器吧！

## B.1 第4章：驾驭Solr的term

第4章重点讲了分析器（analyzers）。让我们来看一下，Solr 是如何允许我们对该章中讲到的分析器进行配置的。分析器将文本：

```
"the doctor's brown fox"
```

翻译成了经过词干处理、不带禁用词的 token：

```
[doctor] [brown] [fox]
```

第4章带给我们的最大收获就是这些 token，它们是在对数据的特征进行建模。控制这一建模过程是管理相关性的基础。同样的内容对 Solr 而言也是 100% 适用的；只是在具体实现细节上有所不同。在本节中，大家首先将会看到，在 Solr 中我们是如何构建一个自定义分析器的。然后将会看到，对于传入 Solr 的文档，我们是如何将分析器映射到文档中的某个具体字段的。

### B.1.1 Solr 的分析与映射功能总结

表 B.1 将一般意义上的分析和映射功能对应到了 Solr 中的相应特性。

表 B.1 Solr 的分析功能

章	功能	Solr 的对应特性
4	自定义分析器	通过自定义 fieldType 在 schema 中实现
4	字段映射（将每个字段都映射到一个分析器上）	通过创建自定义 fieldType 的字段来实现

### B.1.2 在 Solr 中构建自定义分析器

在第4章中，我们看到了 Elasticsearch 是如何利用 JSON 来创建分析器的。这些 JSON 格式的配置是针对某个索引的诸多配置中的一小部分。通常，当创建一个索引时，我们会指定在 Elasticsearch 中所使用的自定义分析器和字段映射：

```
{
  "settings": {
    "analysis": {
      "analyzer": {
```



```
"standard_clone": {
  "tokenizer": "standard",
  "filter": [
    "standard",
    "lowercase",
    "stop"]}]}}}
```

对于 Solr 的读者而言，你是幸运的，我们可以轻易地将本章的代码从 Elasticsearch 翻译到 Solr。大家只需注意一点关键的区别：Solr 用的不是基于 HTTP 的 JSON，它是用 XML 格式在其 schema.xml 配置文件中对分析器进行配置的。

Solr 允许用户定义可对分析进行控制的自定义 fieldType——例如：

```
<fieldType name="text_standard_clone" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StandardFilterFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.StopFilterFactory" words="stopwords.txt" />
  </analyzer>
</fieldType>
```

开始觉得有点眼熟了，是吗？这段分析器的定义对应于之前在 Elasticsearch 里配置的分析器。一旦清楚了对应关系，要在 Solr 中实现第 4 章中的例子就变得很容易了。在前面的片段中，我们用了一个 <tokenizer .../>，将分析器配置成了标准的分词器 (class="solr.StandardTokenizerFactory")。这对应于前面 Elasticsearch 定义中 JSON 格式的元素："tokenizer"。类似的，一行一行的 <filter .../> 指定的是一组待执行的 token 过滤器，就如同 Elasticsearch 的分析器，有一个 filter 元素，后面跟着一个 JSON 列表。

在 Elasticsearch 中，我们是在其他的地方以禁用词列表或文件的形式来配置禁用词过滤器的。这一点 Solr 与之有所不同，它将配置选项放入了过滤器本身的定义中（注意 words="stopwords.txt"）。

Elasticsearch 为分析提供了细粒度的控制，它甚至允许我们在查询阶段指定分析器。Solr 的工作方式有所不同，它只允许我们在 fieldType 这一级进行自定义。fieldType 允许我们定义将分析器用于索引阶段还是查询阶段。因此，如果要在查询阶段使用不同的分析方法，我们可以使用下面的结构：

```
<fieldType name="text_standard_clone" class="solr.TextField">
  <analyzer type="query">
```

```

    ...<!-- query-time analysis -->
  </analyzer>
  <analyzer type="index">
    ...<!-- index-time analysis -->

  </analyzer>

</fieldType>

```

太好啦！我们有一个带自定义分析链的自定义字段类型了。接下来我们要研究如何将其与特定字段关联起来。

### B.1.3 在 Solr 中使用字段映射

Solr 是如何将一个像 title 这样的字段与一个自定义分析器关联起来的呢？在 Elasticsearch 中，分析器是通过映射定义（mappings）指派给字段的。字段映射是通过如下所示的 JSON 语法建立起来的：

```

"mappings": {
  "movie": {
    "properties": {
      "title": {
        "analyzer": "standard_clone"
      }
    }
  }
}

```

Solr 的做法稍有不同，但差别不是太大。在 Solr 中，我们在 schema 中声明字段，通过 type 属性来指定所使用的 fieldType。此处字段 title 的类型是 text\_standard\_clone：

```
<field name="title" type="text_standard_clone"/>
```

Solr 的搜索引擎会对包含 title 字段的文档使用 text\_standard\_clone 索引分析器，将文档的文本进行转换处理。针对 title 字段的搜索，则会使用 text\_standard\_clone 查询分析器，对查询词进行拆分。

如我们所见，除了个别地方需要调整以外，第 4 章的例子是可以直接用于 Solr 的。在实际使用中，两者还有些许不同之处，我们还会发现，各种不同的选项可以从不同方面对设置进行控制。尽管如此，现在大家可以开始使用自己的工具，将文本建模成词（term）了！

## B.2 第5章和第6章：Solr中的多字段搜索

第5章开启了我们的信号之旅。基于 Solr 的实现也是需要信号的。信号将查询阶段的相关性评价映射为对用户有意义的各种因素。查询是关于标题(title)的吗？它在文本中被明确提到了吗？字段成为衡量这些因素的一种单位。在第5章中，为了对信号进行衡量，我们开始精确控制每个字段的构建方法。第6章通过引入以词为中心的搜索这一思路继续着这一话题的讨论。以词为中心的搜索认为，匹配的查询词越多，文档的相关性就越高。

### B.2.1 查询功能的映射总结

表 B.2 将 Elasticsearch 的查询策略对应到了相应的 Solr 策略。

表 B.2 Solr 的查询功能

章	查询策略	Solr 的对应特性
5	best_fields	tie 值为 0.0 的 edismax，接近但不完全等价
5	most_fields	tie 值为 1.0 的 edismax
6	Elasticsearch 中的 query_string	edismax 查询分析器
6	利用 copy_to 实现的自定义全字段	在 schema 中利用 copyField 来构建一个很大的全字段
6	cross_fields	没有对应或接近的特性，但可以利用 Lucene 的 BlendedTermQuery 通过一个自定义的查询解析器来实现

### B.2.2 理解 Solr 和 Elasticsearch 查询的差异

在这个有关查询功能的概述里，大家会注意到，我们把第5章和第6章的内容放在了一起。这是因为，以 Solr 为中心的讨论对于这两章的内容组织方式会截然不同。这是为什么呢？

- Solr 的开箱即用体验不是从以字段为中心的搜索开始的，而 Elasticsearch 却是。
- Solr 是从以词为中心的搜索开始的，并且用到了它的 edismax 查询解析器。而查询解析器在 Elasticsearch 中相对而言并不是那么重要。
- Solr 没有 cross-field 搜索，但我们可以利用查询解析器很容易地将其加入 Solr。

是什么因素导致了这些不同呢？Solr 是一个“富”服务器查询系统（thick

server querying system)；它为我们做了很多工作。而 Elasticsearch，则更多的是一个“瘦”服务器，层次相对少一些。相应的，Elasticsearch 暴露出来的 API 更接近于原生的 Lucene 查询 API，图 B.1 展示了这种不同。

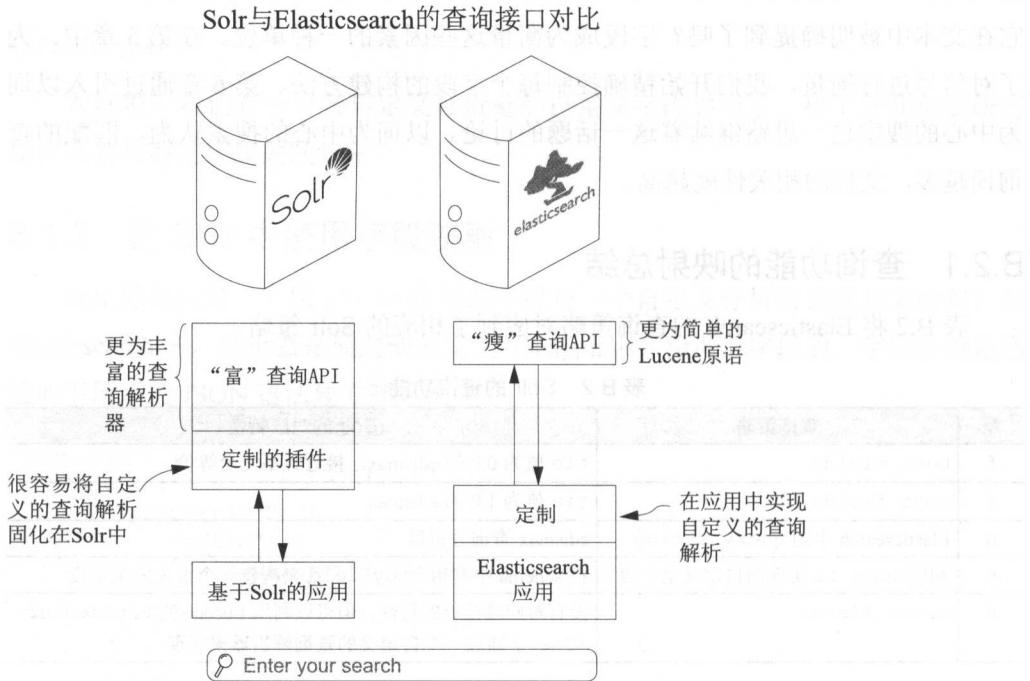


图 B.1 Solr 和 Elasticsearch 中的查询自定义。Solr 使开发者将更多的自定义能力推给了其“富”查询 API。而 Elasticsearch 则暴露出了更为简单的原语，以供开发者使用。

富查询 API 相对于瘦查询 API 意味着什么呢？在查询时，Solr 向富查询解析器的扩展程序库加入了许多功能。查询解析器允许我们利用它自己的自定义查询语法与搜索进行交互。然后再将其翻译成底层的 Lucene 查询，比如布尔查询、词查询，或者 `dismax`。这些查询解析器对许多 Lucene 的细节都做了抽象。Elasticsearch 的策略恰恰相反。它为我们提供了更接近底层 Lucene 查询的原语，以这样的方式来提供服务，使我们能在搜索引擎之外构造出更大、更复杂的查询。Elasticsearch 的查询 API 为用户做得相对少一些。偶有例外的情况也不会有太多的奇迹发生，它不过是将一两个基本设置翻译成 Lucene 的查询罢了（例如，`multi_match`）。唯独 `query_string` 比较特别，它作为一个真正的查询解析器，就像 Solr 中定义的那样。

从很多方面来说，这使得 Elasticsearch 成为大家学习 Lucene 搜索的一个更好的

选择：它的 API 更接近底层的 Lucene 查询。而 Solr 则通常会为我们提供一些功能强大的查询解析器，并屏蔽掉一些具体的细节。不过，Solr 借助底层的 Lucene 代码，使用户能够轻松地创建属于自己的查询解析器。对于 Solr 开发者来说，这也许就是我们开发的第一批插件。查询解析器还有一个蓬勃发展的生态系统，实现的功能包括像图搜索（graph search）或复杂短语搜索（complex phrase search）等。而对于 Elasticsearch 的开发者而言，自定义查询解析器也许是最后一种选择，更有可能的做法则是在搜索引擎之外，通过查询 API 来实现类似的功能。

了解了这些背景信息以后，我们就可以开始研究，Solr 究竟在哪些地方适用于我们在第 5 章和第 6 章中讨论的内容。

### B.2.3 查询 Solr：工效学

作为一名 Solr 开发者，我们知道从工效学的角度查询 Solr 与查询 Elasticsearch 有很大不同。与 Solr 进行交互最为常用的方法是通过基于 HTTP 访问的 URL 参数。关于这一点，我们在第 3 章中指出如何对 Solr 进行搜索时，大家就已经看到了。Solr 的查询是以如下形式来表示的：

```
http://solr.quepid.com/solr/tmdb/select?q=sea biscuit likes to fish!
```

通过对传入 URL 的参数加以控制（很多时候依赖于对当前所用的查询解析器的了解），我们可以对查询字符串如何被翻译成底层的 Lucene 查询进行控制。我们使用 defType 参数实施对查询解析器的控制。此处，我们启用了 edismax 查询解析器，同时还要求通过 qf 参数将可被搜索的字段传入其中：

```
http://solr.quepid.com/solr/tmdb/select?q=sea biscuit likes to fish!&defType=edismax&qf=title overview
```

Solr 还有一种相当特别的语法，称之为局部参数（local params）。这种语法允许我们将参数的范围限定于查询字符串的一部分（因此以局部命名）。前面的 edismax 查询，可以利用如下所示的局部参数的语法，重新进行改写：

```
http://solr.quepid.com/solr/tmdb/select?q={!defType=edismax qf='title overview'}sea biscuit likes to fish!
```

了解了基本的工效学，让我们来看一下 edismax 查询解析器——大部分基于 Solr 的查询方案都以此为起点。看一下它如何适用于我们在第 5 章和第 6 章中讨论

的内容。

## B.2.4 使用 edismax 查询解析器分别以词和字段为中心进行搜索

Solr 时常推荐开发者们从了解 edismax 查询解析器开始，它被称为 Solr 的瑞士军刀。作为一名 Solr 开发者，edismax 是我们相关性工作的核心。edismax 查询解析器所做的工作如下所示：

1. 对查询字符串 ( $q = \dots$ ) 做出评估，判断其是对应于 Lucene 风格的查询字符串，还是常规形式的查询字符串。
2. 如果查询字符串是 Lucene 风格的查询 ( $q=title:"taco nacho"$ )，那就对其进行求值运算。
3. 如果查询字符串不是 Lucene 风格的查询 ( $=sea biscuit likes to fish$ )，edismax 就会执行 dismax 形式的、以词为中心的搜索。
4. 然后逐层进行各种放大处理。

此处，我们将着重研究步骤 3 中的详细情况，看一下如何将其映射到第 5 章和第 6 章中的讨论。值得注意的是，edismax 采用了额外的参数进行加法和乘法放大；在 B.3 节，我们将谈到相关的具体内容。

在第 6 章中我们讲到了，像 `query_string` 这样的查询分析器，如果要对不涉及任何分析器处理的查询文本进行分词操作，就必须要选择一致的方法。edismax 所做的（也是 Solr 对所有查询解析器所做的），就是在进行分析之前按空格对搜索查询进行拆分。因此，edismax 将下面的查询：

```
q=sea biscuit likes to fish!&defType=edismax&qf=title overview
```

转换成了一个逐词进行的 dismax 查询：

```
(title:sea | overview:sea) (title:biscuit | overview:biscuit)
(title:likes | overview:likes) (title:to | overview:to)
(title:fish! | overview:fish!)
```

还记得吗，此处的 `|` 对应于 dismax 操作——取查询评价的最大值。纯粹意义上的 dismax 对应于 `best_fields`，遵循“胜者为王”的策略。评价最高的字段总是会在竞争中胜出。调整该公式的一种方法是调整放大处理。edismax 查询解析器允许我们采用与 Elasticsearch 相同的语法对放大处理进行调整：



```
qf=title^0.1 overview^10
```

大家再回忆一下我们讨论过的 `tie_breaker` 参数，它允许我们将来自于其他“被击败”字段的评价也纳入进来。幸运的是，`edismax` 有一个 `tie` 参数具备同样的功能：

```
tie=1.0
```

在第5章里，我们讨论过将 `tie_breaker` 设置为 1.0 相当于对被搜索字段进行累加求和。在 `Elasticsearch` 的语境里，这样做就是把搜索从基于 `best_fields` 的、“胜者为王”的多字段搜索策略变成基于 `most_fields` 的、“人皆有份”的策略。利用这样的策略，我们让每一个字段的评价在排名函数中都得到了话语权。要复制在第5章中讨论的这种基于 `most_fields` 的多字段搜索，我们只要执行一个 `tie` 取值为 1.0 的 `edismax` 查询就可以了。

## B.2.5 全字段和 `cross_fields` 搜索

要涵盖我们在第5章和第6章里的所有讨论，还需涉及两个主题：`cross_fields` 和全字段。要记住，这两种策略的目标是为了解决字段的不一致问题。默认情况下，评价计算中的 `IDF` 部分是不会跨字段进行计算的。在第6章中我们指出了，像 `edismax` 或 `query_string` 这样的查询解析器，解决了以词为中心进行搜索的一部分问题。但是它们并没有解决字段的不一致性问题。这里我们给出解决字段不一致性问题的两种常用策略：

- 全字段——将多个字段组合成一个字段。这样做就把源字段的文档频率都结合到一起了。
- `cross_fields` 搜索——利用 `Elasticsearch` 的 `cross_fields` 搜索在查询阶段得到跨字段的文档频率，尽管这样做精确性会稍差一些。

### Solr 中的全字段

说到全字段，我们运气很好。正如 `Elasticsearch` 用的是 `copy_to` 一样，`Solr` 的 `schema` 中用的是 `copyField`。利用 `Solr` 中所谓的“拷贝字段 (`copy fields`)”，我们可以达到同样的效果。拷贝字段与字段的其他声明一起，出现在 `schema.xml` 文件中：

```
<copyField source="title" dest="text_all" />
```

此处，`Solr` 在索引阶段将 `title` 字段复制到了 `text_all` 字段。与 `Elasticsearch`

有所不同，Solr 没有内建的 `_all` 字段。但是 Solr 默认的 `schema` 通常会定义一个带有多个 `copy` 指令的 `text_all` 字段。该字段所扮演的角色就相当于 Elasticsearch 中默认的 `_all` 字段。

Solr 中的 `cross_fields` 搜索

对于 `cross_fields` 搜索，Solr 中并没有类似的功能与之对应。但是正如我们所讨论的那样，在 Solr 中实现用户自己的查询解析器是很容易的。其背后的玄机都在 Lucene 的 Java 类 `BlendedTermQuery` 里。该查询类包办了在 Elasticsearch 中有关 `cross_fields` 功能的所有“脏活累活”。如果大家对于在 Solr 中使用该功能感兴趣，我们推荐阅读之前提到的有关 Solr 的链接资源，以实现我们自己的查询解析器。

B.3 第7章：调整Solr的排名函数

第 7 章鼓励大家把相关性看作一种能够被严格加以控制的东西。好消息是，Solr 为我们这张功能对照表配备了充足的“火力”。大家回想一下，第 7 章讨论的重点是放大处理，包括加法放大和乘法放大。我们还讨论了两种放大的模式：函数查询和布尔查询。

Solr，在很大程度上就是通过 `edismax` 查询解析器来支持所有这些功能的。即便最终在易用性方面还有待改进，但 Solr 在这些方面的能力还是超越了 Elasticsearch。

B.3.1 放大功能的映射总结

表 B.3 对一般类型的放大处理与对应的 Solr 特性做了映射。

表 B.3 Solr 的放大功能

章	功能	Solr 的对应特性
7	加法放大，布尔方式	带 <code>bq</code> 参数的 <code>edismax</code>
7	加法放大，函数方式	带 <code>bf</code> 参数的 <code>edismax</code>
7	乘法放大，函数方式	带 <code>boost</code> 参数的 <code>edismax</code>

B.3.2 Solr 的布尔放大

在第 7 章里，我们利用布尔查询实现了加法放大。这种语法使用了 `bool` 查询原语，直接对应于 Lucene 的布尔查询，如下面的查询片段所示：

```

"query": {
  "bool": {
    "should": [
      {
        "match": {
          "title": "Rambo"
        }
      }
    ]
  }
}

```

采用加法放大的布尔查询

Solr 的“布尔”放大，其效果与之类似。就像用 Elasticsearch 一样，可以对任何我们喜欢的查询进行放大，只是语法看起来完全不同。在 Solr 中，我们通过参数 `bq` 实现对查询的放大。回忆一下，Solr 有一个被称为局部参数的查询语法，允许我们指定所使用的查询解析器。加法放大是通过带一个加法放大参数 `bq` 的局部参数语法来实现的，例如：

```
bq={!defType=lucene}title:Rambo
```

此处，我们通过 Lucene 的查询解析器，执行查询字符串 `title:Rambo`，以此来告知 Solr 进行加法放大。该查询解析器的输出结果将会作为一个布尔型的 `SHOULD` 子句，被追加到 `edismax` 基准查询的后面。对 Solr 来说，将多个查询解析器累加起来的做法丰富了它的功能。Solr 在功能上如此丰富的表现，使我们能够将各种各样的定制功能层层纳入放大处理的进程之中。

### B.3.3 Solr 的函数查询

正如 Elasticsearch 那样，Solr 也允许我们执行加法或乘法形式的函数查询。在 Elasticsearch 中，我们采用的是以 JSON 格式定义的一个函数列表的形式。例如，针对 TMDb 影片的评价 (`vote_average`)，考虑下面这样一个简单的放大处理：

```

"query": {
  "function_score": {
    "query": {
      ...
    },
    "functions": [
      {
        "field_value_factor": {
          "field": "vote_average",
          "modifier": "sqrt"
        }
      }
    ],
    "boost_mode": "sum"
  }
}

```

主搜索查询

对 `vote_average` 求平方根

将这些函数的计算结果和主查询的评价值相加

在 Solr 中，其语法看起来更像是一个 Excel 电子表格中的公式。而且，Solr 还有一个非常强大的附加功能是 Elasticsearch 非常欠缺的，那就是：查询函数。是的，可以把查询函数添加到我们的函数查询中——把这句话快速重复三遍吧！<sup>1</sup> 查询函数允许我们执行任意形式的查询语句，并将它的  $TF \times IDF$  相关性评价注入函数查询本身（就如同此前在 Elasticsearch 的代码片段中，我们在一个函数列表中指定的一样）。

那么，Solr 的函数查询长什么样呢？正如我们之前所说的，它看起来就像是直接从电子表格里取出来的一样。edismax 的 bf 参数执行的是一个加法形式的函数放大。下面是一个按数值型字段 popularity 执行加法放大的函数查询：

```
bf=sqrt(popularity)
```

或者，用 title 字段里针对 rambo 的  $TF \times IDF$  评价乘以 popularity 的平方根，我们也许会采用如下形式的查询函数：

```
ramboScore={!defType=luene}title:Rambo  
&bf=product(sqrt(popularity),query($ramboScore))
```

最终我们得到了一个完整的 Solr 查询，包含了所有的参数，如下所示：

```
http://solr.quepid.com/solr/tmdb/select?q=*&defType=edismax  
&qf=titleoverview&ramboScore={!defType=luene}title:Rambo  
&bf=product(sqrt(vote_average),query($ramboScore))
```

此处，我们把放大处理分成了两个部分：一个查询和一个函数。对于查询，大家可以看到，Solr 允许我们用变量来组织各个组成部分。这一点可以通过前面的 ramboScore 变量看到。<sup>2</sup> 该查询在 title 字段中针对“Rambo”执行了一次简单的搜索。函数 bf 对电影的受欢迎程度取平方根，并乘以电影片名针对“Rambo”的  $TF \times IDF$  评价。然后将得到的结果加入整体的相关性评价（即加法放大）。最终

---

<sup>1</sup> 查询函数和函数查询容易混淆，作者在此处是玩笑的口吻，有“绕口令”之嫌。——译者注

<sup>2</sup> 要了解更多有关 Solr 放大处理的组织方法，我们推荐由本书作者之一所写的这篇文章：*Parameterizing and Organizing Solr Boosts*，该文章发表在 OpenSource Connections 的博客上，<http://opensourceconnections.com/blog/2013/11/22/parameterizing-and-organizing-solr-boosts/>。对于像第7章中我们所讨论的那些策略，涉及专门针对 Solr 的部分，我们推荐 OpenSource Connections 博客上的 *Improve search relevancy by telling Solr exactly what you want* 这篇文章，<http://opensourceconnections.com/blog/2013/07/24/improve-search-relevancy-by-telling-solr-exactly-what-you-want/>。

得到的结果是：广受欢迎的“Rambo”系列影片被排到了搜索结果的最前面。

正如我们所看到的，Solr 的函数查询远比 Elasticsearch 的简练。但是我们可能也已经看到了，如果达到足够的复杂度，它们就会变得很难操控！在这个方面，Solr 可能有点像 Perl；支持它的人已经学会用它那强大的威力去做一些不可思议的事，而反对它的人则看到了这样的函数查询带给我们糟糕的可读性，并对其心生畏惧。

Elasticsearch 允许我们在 `function_score_query` 中以一个函数的形式来编写自定义的脚本。脚本在 Elasticsearch 中可以用多种编程语言来实现，比如 Groovy 和 JavaScript。一个真正的编程环境，其优势就在于它具有好的可读性。

### B.3.4 Solr 中的乘法放大

Elasticsearch 的乘法放大可以通过在前面提到的 `function_score_query` 中把累加求和改为乘法来实现。类似的，在 Solr 中使用乘法放大需要对加法放大做一点简单的修改。

Solr 的乘法放大是通过函数查询的执行结果来完成的，它用 `boost` 参数乘以作为基准的相关性评价价值。例如，为了让结果偏向“受欢迎程度（popularity）”，我们也许可以用下面的方法来实现乘法放大：

```
boost=sqrt(popularity)
```

## B.4 第8章：相关性反馈

第 8 章讨论了用户体验的各种要素及其重要性，比如：切面（`faceting`）、自动补全（`autocomplete`）、高亮显示（`highlighting`），以及字段合并（`field collapsing`）。在那一章里，这些用户体验的实现是通过我们在前几章中讨论的许多相关性功能来完成的。不过，我们的确也讨论了一些前几章里没有介绍的功能。因此，为了帮助大家完成对第 8 章中剩余内容的考察，我们将通过对照“功能百宝箱”中的功能，继续充实有关于 Solr 的讨论。

### B.4.1 相关性反馈的功能映射总结

表 B.4 把 Elasticsearch 的反馈功能映射到了 Solr 中的对应特性。

表 B.4 Solr 中的相关性反馈功能

章	功能	Solr 的对应特性
8	切面（在 Elasticsearch 中通过聚合来实现）	Solr 的切面
8	短语前缀查询	Solr 的 complexphrase 查询解析器
8	分组（在 Elasticsearch 中通过 top_hits 聚合来实现）	Solr 的 field-collapsing 功能
8	建议和拼写检查	请参考 Solr 有关建议和拼写检查功能的文档
8	高亮显示	请参考 Solr 有关 highlighter 的文档

## B.4.2 Solr 的自动补全：匹配短语前缀

在 Elasticsearch 中，实现自动补全方案的一种策略是依赖于 `match_phrase_prefix` 查询。该查询会针对最后一个单词执行一条带前缀的短语查询，如下列代码片段所示：

```
{ "query": {
  "match_phrase_prefix" : {
    "title": "star tr"}}
```

Solr 在其 `complexphrase` 查询解析器中提供了类似的功能。该查询解析器可以将一个以 “\*” 结尾的字符串当作 `match_phrase_prefix` 来解释。因此，前面的查询就变成了 `title:star tr*`，如下所示：

```
http://solr.quepid.com/solr/tmdb/select?q=title:star
tr*&defType=complexphrase
```

该 Solr 查询执行了一条等价于 `match_phrase_prefix` 的短语查询，重点关注的是短语，直至短语中的最后一个单词。结尾处的 “\*” 告诉查询解析器，要针对最后一个单词执行一次前缀搜索（`prefix search`）。

## B.4.3 Solr 中的切面浏览

切面是绝大多数搜索和浏览应用至关重要的组成部分！它把搜索结果分成了一系列可供过滤的类别。第 8 章使用了 Elasticsearch 的 `term` 聚合来实现切面搜索和某些自动补全功能。参见下列代码片段：

```
"aggregations": {
  "completion": {
    "terms": {
      "field": "title"}}
```



使用 Solr 的切面功能，我们会获得类似的能力。前面谈到的聚合可以用一个针对 title 字段的简单切面来实现：

```
facet=true&facet.field=title
```

类似的，Elasticsearch 允许我们使用 include 参数来包含 / 排除聚合的结果，如下列代码片段所示：

```
"terms": {
  "field": "title",
  "include": "rambo*",
}
```

Solr 通过 facet.prefix 选项提供了同样的能力：

```
facet=true&facet.field=title&facet.prefix=rambo
```

把前述内容结合起来，完整的切面查询就变成了下面的样子：

```
http://solr.quepid.com/solr/tmdb/select?q=*&facet=true&
facet.field=title&facet.prefix=rambo
```

现在我们可以开始使用切面功能了！

## B.4.4 字段合并

第 8 章提到了，我们常常会希望对搜索结果以层级结构的方式进行分组。Elasticsearch 是通过其 top\_hits 聚合来支持这一功能的。这种方法通过一个常见的公共属性对搜索结果进行分组。例如，考虑下面这个查询：

```
"aggs": {
  "original_versions": {
    "terms": {
      "field": "original_id",
      "order": { "top_score": "desc" },
      "aggs": {
        "hits": {
          "top_hits": { "size": 1 },
          "top_score": {
            "max": { "script": "_score" }
          }
        }
      }
    }
  }
}
```

按 original\_id 分组

根据每个全局唯一的 original\_id 实现的 top\_hits 子聚合

该查询按照 original\_id 对结果进行分组，然后按 original\_id 分组显示评价。Solr 也能做到这一点！Solr 有一个被称为 field-collapsing 的功能，通过其 group 参数实现了同样的功能：

```
&group=true&group.field=original_id&group.limit=3
```

这样的查询，其返回的结果会按 `original_id` 进行分组，在本例中，每个唯一的 ID 会有三条结果。我们还可以在 Solr 的文档中查阅其他与分组相关的参数。

### B.4.5 建议和高亮显示

拼写检查、查询建议，以及高亮显示对任何一种搜索应用而言都是有很大帮助的。第 8 章我们展示了 Elasticsearch 的建议和高亮显示功能。因为我们讨论的主题都是和支持相关性有关的，所以此处我们不会对两个搜索引擎的拼写检查和高亮显示模块进行详细对比。

相反，我们只是告诉大家，Solr 中也存在几乎相同的功能。Solr 有一个预置的建议模块和一个拼写检查模块。它还自带一个高度可配置的（并且是可插拔的）高亮显示模块。在大多数情况下，利用默认的设置，我们就可以通过 `suggest=true` 和 `hl=true` 来开启这些功能。例如，下列查询的结果中包含了高亮显示的内容：

```
http://solr.quepid.com/solr/tmdb/select?q=sea biscuit likes to fish!&hl=true
```

这些模块包含了大量丰富的可用功能。我们邀请大家访问 Solr 的官方文档，从中探寻这些模块的各种能力。

# 相关性搜索

用户期望并习惯于得到即时而相关的搜索结果。要做到这一点，我们就必须熟练掌握搜索引擎的工作原理。然而对于许多开发者来说，相关性排名神秘莫测，令人一头雾水。

本书揭开了这一主题的神秘面纱，并告诉大家搜索引擎就是一个可编程的相关性框架。利用 Elasticsearch 和 Solr，本书还讲述了如何在该框架中表达业务排名的规则。我们将学会如何对相关性进行编程，以及怎样结合各种次级数据源、分类方法、文本分析手段，还有个性化需求。实际上，相关性框架也需要一定的软性技能，比如，与利益相关者协作，为业务找到正确的相关性需求。最终，我们将能够在搜索产品的整个生命周期中，实现相关性改进的一个可证明、可度量的良性循环。

## 本书包含的内容：

- 相关性调试技术
- 将搜索引擎的诸多特性应用于实际问题中
- 利用用户界面来引导搜索的使用者
- 一套针对相关性的系统化方法
- 一种致力于提高搜索质量的企业文化

本书适用于那些想利用 Elasticsearch 或 Solr 尝试构建智能搜索应用的开发人员。

Doug Turnbull 是 OpenSource Connections 搜索相关性业务的首席顾问，在那里他经常发表观点，更新博客。

John Berryman 是 Eventbrite 的一名数据工程师，在那里他专攻推荐和搜索。

这是我所读过的最优秀、最引人入胜的技术书籍之一。

—引自 *Solr in Action* 的作者  
Trey Grainger 为本书撰写的序言

本书将帮助你解决真实世界中基于 Lucene 的搜索相关问题。

—Dimitrios Kouzis-Loukas,  
彭博资讯

这是一本鼓舞人心的书，它揭示了相关性搜索的本质和机理。

—Ursin Stauss, 瑞士邮政

本书以无价的知识武装读者，并利用现代化搜索引擎提供的强大功能，对搜索结果的相关性进行调试。

—Russ Cam, Elastic

MANNING



策划编辑：许 艳  
责任编辑：刘 舫  
封面设计：吴海燕

上架建议：搜索引擎

ISBN 978-7-121-32721-6



9 787121 327216 >

定价：99.00元